UNIVERSITAT POLITÈCNICA DE CATALUNYA

Desenvolupament d'un model de simulació local de moviment de personatges virtuals

Memòria final

Facultat d'Informàtica de Barcelona

Autor: Alejandro Martínez Zamorano

Director: Alejandro Ríos Jerez

Codirectora: Dra. Nuria Pelechano Gómez

Vull agrair la gran ajuda i suport que m'han donat els professors Alex Rios i Nuria Pelechano a l'hora de dur a terme aquest projecte, al meu company de feina Jesus Rodríguez per haver-me donat tantes bones idees que m'han estalviat tantes hores de feina, al meu Team Lead Ricard Fusté per haver-me escapat tants dies abans de la feina però sobretot a la meva parella Marina Garrido per haver-me aguantat tot aquest temps i per haver-me donat tant suport durant aquests mesos.

Índex de contingut

Índex de taules i figures

0 Resum / Abstract

Per començar, en aquest estudi analitzarem el comportament del motor de videojocs [1] *Unity* [2] en relació als algorismes que implementa sobre esquivar objectes, seguir a un líder, interacció entre multituds de personatges virtuals...

Seguidament veurem les mancances que hi ha actualment en aquest software, arxiconegut i molt utilitzat tant per desenvolupadors com per grans empreses com Nintendo [3], Niantic [4] i moltes altres.

A continuació, descriurem diferents estudis actuals que parlen sobre aquest tema i els implementarem, veient així la diferència entre abans i el després. Ens centrarem sobretot en diferents estudis d'universitats i grups d'investigació que tracten aquest tema i bones maneres de resoldre'l.

Finalment, el que es pretén amb aquest treball és la millora de certs algorismes que disposa *Unity* per defecte i que són bastant millorables donat que són poc realistes, tenen comportaments estranys, etc.

In this project we will analyse the behaviour of several algorithms of the game engine Unity, such as collision avoidance, follow a leader, interaction between crowds, etc.

On the one hand, we will see some lacks of this software, very used by several companies, lots of individual developers, investigators...

On the other hand, we will describe different current studies that talk about this topic and we will implement them, seeing the difference between before and after. We will focus mainly on different studies of universities and research groups that deal with this issue and have good ways to solve it.

Finally, the main goal of this project is the improvement of certain algorithms used by Unity by default and that are quite improvable because they are unrealistic, have strange behaviours, etc.

1 Context

1.1. Introducció

La informàtica tal com la coneixem, avança a passos de gegant en el temps. Des de pràcticament sempre, s'ha complert el que es coneix com a Llei de Moore [5], que resumint-la diu que cada 2 anys es duplica el nombre de transistors d'un processador. Això vol dir que en pocs anys ha augmentat d'una manera increïble la capacitat de càlcul dels ordinadors, això ens permet fer càlculs científics d'una manera que anys enrere eren senzillament impossibles, jugar a jocs que no són gaire diferents de la vida real o inclús dotar d'intel·ligència i capacitat de raonament a les màquines. Només cal pensar en com era la vida fa 10 anys. Una cosa pràcticament imprescindible avui en dia és el *smartphone* i només fa 10 anys que va sortir el *iPhone*, que va ser una revolució al mercat.

Com hem comentat anteriorment, les branques que han sortit molt beneficiades d'aquest augment de potència són la branca de la informàtica gràfica i la d'intel·ligència artificial, de les quals, afortunadament, pertany el nostre treball.

L'objectiu principal del projecte és, com ja bé diu el títol del projecte, un simulador de moviment de múltiples personatges virtuals, és a dir, davant d'un escenari que pot ser qualsevol cosa, un simple pla, un bosc, una ciutat, afegir diferents personatges virtuals i veure la interacció tant entre ells, com amb l'entorn, ja sigui esquivant obstacles fixos, esquivant personatges virtuals, canviant forces, acceleracions, distancies, etc.

Unity té instal·lat per defecte algorismes per a esquivar objectes, trobar camins mínims, i d'aquest estil, però no són realistes perquè en molts cops realitzen moviments bruscos, no esquiven obstacles fins que no és massa tard, hi ha comportaments estranys, etc. La nostra idea és millorar aquests algorismes i que qualsevol persona ho pugui utilitzar.

1.2. Actors implicats

Com a *stakeholders* del projecte, és a dir, tothom que té relació amb ell, sigui com a client final o sigui com a desenvolupador amateur, passant pels desenvolupadors, arquitectes, directors del projecte, etc. Donat aquestes premisses, podem identificar amb seguretat els següents actors implicats:

1.2.1 Director, codirectora i desenvolupador

En el nostre cas, aquestes tres figures són les encarregades de portar a bon port aquest treball, ja sigui dirigint el fil del treball, com ajudant a aconseguir els objectius. Com a director del treball tenim al Professor Alejandro Ríos i com a codirectora, a la Nuria Pelechano. Els dos tenen una dilatada experiència en temes de computació gràfica, així que són els més indicats en portar a terme aquest treball. Com a desenvolupador, tenim a l'Alejandro Martínez, que serà qui implementi la solució, així com el mateix *Testing* i *Building* de la solució.

1.2.1 Empreses de videojocs

Ja sigui en l'àmbit amateur com més professional, tenir clar el comportament de molts personatges virtuals a la vegada i la seva interacció, cerca de camins mínims, esquivar obstacles, etc., és clau per poder definir i implementar correctament la majoria de jocs d'avui en dia, sobretot els coneguts com a Estratègia en temps real [6] com l'arxiconegut, *Age of Empires* [7].

1.2.2 Usuaris de Unity

Està pensat per usuaris amb inquietuds que vulguin testejar com es comporten diferents models en diferents situacions, canviar les variables, afegir o treure obstacles, etc. I òbviament, si vol afegir noves funcionalitats o millorar el codi, és totalment obert, així que es pot modificar o estendre sense cap problema.

2 Estat de l'art

Ens podem remuntar molt enrere tractant aquest tema. Tan enrere com els mateixos ordinadors. El mateix John Von Neumann [8] va proposar un problema l'any que intentava trobar una màquina hipotètica que pogués construir còpies de si mateixa i va tenir èxit quan va trobar un model matemàtic per a una màquina amb regles molt complexes en una graella rectangular. Això és una mica complicat d'entendre, però gràcies a això, John Horton Conway [9] l'any 1970, va crear el famós Joc de la Vida [10].

Aquest joc és molt interessant donat que és equivalent a una màquina universal de Turing [11], és a dir, que tot el que es pot computar algorísmicament es pot computar al joc de la vida. Això és molt interessant per a molts camps de la ciència com a les matemàtiques, economistes, etcètera, ja que poden observar visualment com patrons complexos poden provenir de la implementació de regles molt simples.

No entrarem en detall en que consisteix el Joc de la Vida, però el que es interessant per nosaltres es que utilitza la computació gràfica per a simular diferents patrons de models i una primitiva interacció entre ells. Recordem que parlem dels anys setanta.

Des d'aleshores, aquest camp no ha deixat de créixer. Tenim exemples interessants com el que va fer Craig Reynolds' a finals dels 80, on el seu model matemàtic ha contribuït al desenvolupament dels primers models biològics basats en agents on contenen característiques social com a primer intent de modelar la realitat d'agents biològics. D'aquí es on apareix per primer cop el terme vida artificial [12].

Més recentment s'han desenvolupat models matemàtics avançats on simulen la mateixa interacció humana entre individus i hi ha molts grups d'investigació que estan fent recerca en aquest camp donada la importància per a molts camps de la ciència, com pot ser a la psicologia, medicina o inclús amb el màrqueting, donat que hi ha estudis en el que s'estudien els hàbits del consum de les persones i la seva interacció amb diferents productes [13]. També recentment s'ha analitzat en temes de propagació de malalties [14], migracions forçades de persones [15], ecologia [16], congestions de tràfic [17], aplicacions militars [18]...

El que pretenem amb aquest treball no és la interacció a aquest nivell, sinó més en l'àmbit gràfic de com actuen uns personatges virtuals en un escenari. En aquest subconjunt del camp, que ja hem vist que és molt ampli i abasta molts temes diferents, trobem treballs i investigacions en els que començarem a estudiar-los i aplicar-los al nostre simulador a fi que els podem testejar i avaluar, per tant, diguem que aprofitarem el que està fet el dia d'avui i l'ampliarem i l'adaptarem per utilitzar-ho al nostre simulador.

Un gran començament és el article de Craig Reynolds [19], abans també hem mencionat aquest autor, en el que introdueix com treballar amb personatges virtuals autònoms, animacions i jocs. Presenta solucions de com interactuar amb el món virtual d'una manera realista, com esquivar obstacles, agafar el camí més curt entre dos punts, seguir a un altre personatge, unir-se a un grup de personatges, etc.

També investigarem sobre treballs publicats per la Universitat de Carolina del Nord, com poden ser aquest article [20], on es proven i simulen diferents patrons de multituds utilitzant un model anomenat Síndrome d'Adaptació General, on simulen multituds de persones i imiten en certa mesura el comportament humà, com pot ser esquivar objectes en moviment, córrer si s'apropa algun perill o pànics, com pot ser una evacuació per un incendi, deixar distància de seguretat entre persones, etc.

El nostre treball es fonamentarà en aquests articles i en d'altres més avançats que anirem buscant i investigant a mesura que avanci el nostre treball.

3 Formulació del problema

3.1 Problema

Volem fer un model de simulació en el que tenim un nombre determinat de personatges virtuals que variarà, podent així augmentar o disminuir la complexitat de la simulació, on aquests personatges són independents entre si, és a dir, amb la seva pròpia intel·ligència artificial, però que a la vegada podran interactuar entre ells.

Amb un exemple s'entendrà de seguida: Un banc de peixos. Els peixos en si són independents entre ells, però al final tots tenen un mateix objectiu comú, la supervivència, i això ho aconsegueixen actuant com si fossin un, junts però sense col·lisionar, nedant estant units però canviant de direcció i velocitat ràpidament si la situació ho requereix.

La idea és simple però amaga una complexitat elevada. Existeixen molts algorismes, mètodes i models matemàtics però que estan per separat. Molts estan en diferents llenguatges i per diferents plataformes. La nostra idea és fer un *framework* comú on siguem capaços d'agrupar totes aquestes idees en un repositori comú per tal de que sigui accessible, extensible i usable per tothom.

3.2 Objectius

Per a portar a terme el treball i poder treballar en el problema abans mencionat, hem definit uns objectius que hem de complir al llarg del treball, ja que ens ajudaran a assolir els requisits necessaris que s'espera del nostre projecte:

- Estudiar a fons el comportament de Unity respecte els algoritmes que utilitza i que volem millorar per saber realment el que volem millorar. Saber com estan implementats internament.
- Millorar els algorismes mencionats per tal de millorar el realisme sense posar en perill la eficiència.
- Penjar el nostre codi a la web per que qualsevol pugui implementar aquestes millores per tal de que es puguin reaprofitar.

3.3 Aportació de l'especialitat de Computació

3.3.1 Competències transversals i nivell d'assoliment

- **CCO1.1:** Avaluar la complexitat computacional d'un problema, conèixer estratègies algorísmiques que puguin dur a la seva resolució, i recomanar, desenvolupar i implementar la que garanteixi el millor rendiment d'acord amb els requisits establerts. **[En profunditat]**

  Aquest requisit ve donat clarament perquè s'han d'analitzar algorismes existents de *Unity*, veure quin és l'espai de millora i implementar-ne de nous amb la màxima eficiència possible. En gran mesura, aquest requeriment és el més bàsic en el que podem basar-nos per a la realització del treball. Es preveu assolir el nivell de coneixement, ja que el nostre treball compleix punt per punt tot el que diu el requeriment.

  Concretament s'ha implementat un algorisme de *Collision Avoidance* i *Obstacle Avoidance* que permet el moviment realista i amb suavitat dels personatges i s'ha hagut de tenir molt en compte la seva eficiència per a què no afectés el rendiment de l'aplicació.

- **CCO1.3:** Definir, avaluar i seleccionar plataformes de desenvolupament i producció hardware i software per al desenvolupament d'aplicacions i serveis informàtics de diversa complexitat. **[Bastant]**

  Donat que necessitem utilitzar software complexa com és el Visual Studio, *Unity*, *Team Foundation Server* per a temes de *Building i Testing i control de versions*, havent també avaluat els motius d'utilitzar una o una altra plataforma. El nivell de coneixement s'assolirà tenint expertesa en la decisió i la utilització d'aquest software i els motius per fer-ho.

- **CCO2.6:** Dissenyar i implementar aplicacions gràfiques, de realitat virtual, de realitat augmentada i videojocs**. [En profunditat]**

  Aquest requeriment també és bàsic per a la realització del treball, ja que com bé diu, dissenyarem i implementarem una aplicació de gràfica i de videojocs, donat que estem alterant el comportament que ve per defecte en *Unity* per a millorar-lo en el sentit comentat anteriorment de les aplicacions gràfiques, com poden ser simuladors de multituds o el món dels videojocs.

3.3.2 Justificació de l'especialitat

Com hem pogut veure en els apartats anteriors, el que es pretén fer és l'estudi de publicacions de temes d'algorísmia i gràfics, la millora d'algorismes i la implementació i verificació de la correctesa del que s'ha fet. Això clarament té un marcat caràcter de l'especialitat de computació.

Tot el tema de gràfics, complexitats, algorísmia, interacció d'individus d'una manera intel·ligent, etc. Tot això ve donat per les assignatures impartides a l'especialitat on es fa èmfasi en l'eficiència i la rugositat del codi. També proporciona la capacitat d'aquest pensament crític tan necessari quan el domini és tan concret i complexa i s'ha d'entendre en tot moment on s'està i on es vol anar.

Si bé és cert que només a l'especialitat d'Enginyeria del Software es treballen temes de metodologies de treball (com Scrum) o verificació i validació del codi mitjançant testos, molt presents en el nostre treball i en el que no he rebut cap coneixement des de la universitat i, personalment penso que tot enginyer informàtic hauria de saber sent assignatures obligatòries troncals, l'especialitat de computació ens dóna la visió i la capacitat de cercar informació i aprendre a treballar d'una manera autònoma per tal de solucionar les mancances que es poden tenir en altres àrees.

3.3.3 Assignatures de computació relacionades

Algorísmia: Assignatura clau en aquest projecte donat que allà vam obtenir un coneixement profund i robust dels algorismes clàssics i com aconseguir adaptar-los, o si més no, adaptar la mateixa idea per a solucionar problemes que a simple vista, no tenen una semblança o uns patrons comuns. També grans coneixements sobre la complexitat dels algorismes i estructures de dades i com saber en tot moment que és millor utilitzar.

Gràfics: Gràfics ens aporta el coneixement necessari de tot el càlcul algebraic tan necessari en la computació gràfica, així com el càlcul de trajectòries de llum, moviments, escalats, rotacions, etc. Si bé es cert que molt d'aquest coneixement ve donat per les assignatures de M1 i M2, aquí s'ha utilitzat aquest coneixement d'una manera pràctica.

Intel·ligència artificial: Aquesta assignatura ens aporta tot el coneixement que ens calen al projecte sobre algorismes com el A*, massivament utilitzat en la gran majoria de jocs, tant antics com actuals. També un gran coneixement sobre d'altres algorismes de cerca local, ja que normalment s'utilitzen variants del A* amb algorismes de cerca locals perquè la complexitat no es dispari. També ens aporta la idea dels sistemes basats en el coneixement per a tenir una idea de com s'hauria de fer la interacció amb l'entorn tenint un bon coneixement i molta informació del domini.

4 Abast

4.1 Abast del projecte

L'objectiu principal d'aquest treball, és fer el simulador que hem comentat, provant i testejant diferents algorismes existents i penjar-ho a la web per tal que sigui accessible per tothom qui el vulgui fer servir.

Definirem els objectius principals i els passos a seguir per al compliment dels requisits del treball:

- Estudiar els diferents algorismes i models matemàtics ja existents a fi que descobrim on està la frontera entre el que està implementat i el que ens agradaria implementar. Descobrir també les limitacions i punts forts de cada model estudiat per decidir si cal o no implementar-lo a la nostra solució, és a dir, si l'algorisme en concret ens aporta valor o hi ha alguna alternativa millor.
- Aprenentatge i utilització del programari *Unity* i integració d'aquest amb els models i algorismes anteriorment estudiats fent èmfasi en l'eficiència i el rendiment de la solució donat que seran algorismes costosos i càlculs complexos. *Testing* i *building* de la solució en servidor de control de versions *(Team Foundation Server).*
- Verificació de la nostra solució per comprovar que els resultats de qualitat són els esperats, tant per part del desenvolupador del projecte com per part dels directors d'aquest. Comprovació que tots els requisits inicialment definits s'han cobert i estan implementats i testejats.
- Publicació del nostre codi a la web perquè qualsevol persona o organització que vulgui, el pugui utilitzar o estendre la seva funcionalitat.

4.2 Obstacles i riscos del projecte

4.2.1 Desconeixement de les tecnologies utilitzades

Donat que a la carrera no hem vist gran part del contingut d'aquest treball, és un començament complicat, ja que hem d'aprendre primer, com funciona el motor de videojocs *Unity*, el qual és complexa i molt potent. També estudiar i investigar els algorismes i models que pretenem utilitzar en el nostre simulador ja que són complexos i sensibles a errors que puguem introduir a l'hora de codificar-los.

4.2.2 Restricció temporal

Aquest treball té una duració de 4 mesos, dels quals no són a temps complet donat que el desenvolupador té una feina a mitja jornada i d'altres classes a la universitat, per tant el temps és un handicap important. S'ha d'aprofitar bé i tenir molt clar el que es vol fer en tot moment a fi que no es perdi el focus i objectiu final d'aquest treball.

4.2.3 Depuració i testeig

Aquest factor serà clau per l'obtenció dels resultats esperats, ja que amb errors que puguem introduir o ja existents, podem obtenir solucions errònies o d'una qualitat que no és l'esperada en un treball de fi de grau, per tant s'ha d'anar amb compte i fer un testeig exhaustiu de la solució, el qual tampoc serà gens fàcil donada la naturalesa complexa dels algorismes que pretenem utilitzar. També s'ha de realitzar un testeig visual exhaustiu per tal de trobar comportaments estranys o no desitjats.

4.2.4 Eficiència

L'eficiència serà un factor determinant atès que els algorismes i models matemàtics a utilitzar tenen un cost computacional alt i si no ho tenim en compte, se'ns pot disparar la complexitat i fer la solució inabordable.

5 Metodologia i rigor

Utilitzarem una metodologia àgil que utilitza el framework de Scrum [21]. La principal característica que té és que segueix un model iteratiu i incremental. Utilitzarem com a mètode iteratiu els Sprints, que tenen una duració de dues setmanes, és a dir, el primer dia del *sprint*, definirem les tasques per a dues setmanes vista i a la finalització del *sprint*, farem una revisió i una presentació de les tasques fetes durant aquelles setmanes, si s'han assolit, si s'ha tingut problemes, etc.

En Scrum és molt important l'estimació de les tasques que es volen fer durant la iteració, per tant el primer dia del *sprint*, els directors i el desenvolupador faran una estimació en hores per tal que la quantitat de feina sigui l'adequada. Aquestes tasques quedaran reflectides en el *Sprint Backlog*, que és una espècie de tauler on estan els Product Backlog Items, que són les tasques més genèriques, i dintre estan les tasques més petites. Per exemple, un PBI pot ser Implementar l'algorisme de Reynolds. Això és una tasca genèrica, dins d'ella podem trobar tasques petites com cerca d'informació sobre l'algorisme, implementar la variant X, implementar la variant Y, building i testeig de la solució, etc.

ID	Title	Effort	Remaining...	Story Points	State
713748	◢ [TechnicalDebt] Treating Warnings as Errors in CP projects			5	Ready for t...
713150	Track Warnings that cannot be removed.	2	0		Done
713151	Fix ME Code Warnings		0		Done
714330	Investigate why ME CsvToXml project raises an exception in the build		0		Done
708797	Enable TreatWarningsAsErrors in all projects	4	0		Done

Figura 1: Exemple de Backlog.

En *Scrum* també es fa una reunió diària amb els membres de l'equip, la qual sol durar entre 10 i 15 minuts on s'explica a l'equip que es va fer el dia anterior i el que es pretén fer durant el dia actual. Donat que no som un equip *Scrum* com a tal, intentarem tenir una comunicació fluida mitjançant el correu electrònic i petites reunions durant *l'sprint*.

Pel tema de l'Entrega Continua / Integració Continua, que resumidament, és que cada release de cada *Sprint* de la nostra solució serà funcional, ja que tindrem una bateria de testos que provaran que la nostra solució sigui vàlida en tot moment, utilitzarem Team Foundation Server de Microsoft [22] que és un servidor de control de versions on podrem executar en un servidor remot els nostres testos i obtenir feedback d'ells si hi ha algun error.

La gràcia d'això és que tindrem un històric de tots els canvis de codi que hem fet amb el seu corresponent resultat de l'execució i tot això serà automàtic, cada cop que es faci un *check-in* del codi, automàticament es dispararà una *Build Definition* que farà la compilació i execució dels testos en remot. Si l'execució d'aquesta *Build* ha sigut correcte, el codi serà pujat al repositori, si hi ha algun problema, no hi haurà cap canvi al repositori. Així ens assegurem que el codi que hi ha a la branca és correcte. Això es conegut com a Gated Check-In.

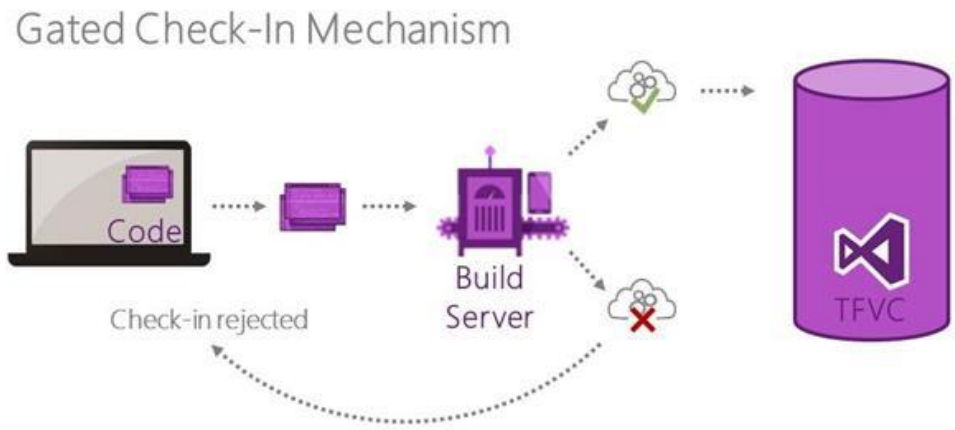


Figura 2: Funcionament del Gated Check-In

Un gran avantatge d'això és que la compilació i execució dels testos serà en remot, per tant no ens consumirà recursos propis i podrem seguir treballant normalment i sense interrupcions, ja que la compilació de la solució i l'execució dels testos sol tenir un fort impacte a la CPU de l'equip. A més de tenir la comoditat de que allà existeix el històric de compilacions i execucions i es pot veure durant tot el projecte com ha anat avançant.

Com a màquina que executa el nostre codi hem utilitzat un altre ordinador personal que ha fet de servidor, però també existeix la possibilitat en fer-ho a una maquina gratuïta proporcionada per *Microsoft*, l'únic que a vegades s'ha d'esperar una mica perquè s'ha de fer una "cua" perquè pot haver mes usuaris que l'estiguin utilitzant. També existeix la possibilitat de fer-ho de pagament i tenir una màquina pròpia muntada al *Cloud* de *Microsoft* anomenat *Microsoft Azure* [23].

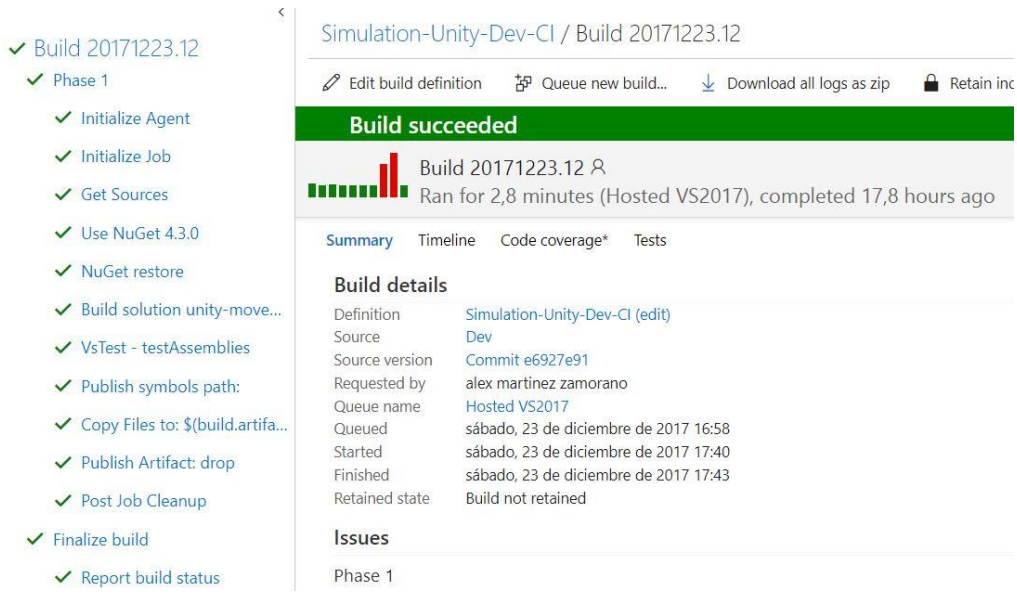


Figura 3: Exemple d'execució d'una build en TFS

Tot i que hem tingut diversos obstacles al camí, al seguir una metodologia Scrum amb sprints curts, s'han realitzat reunions sovint i d'aquesta manera s'ha evitat un augment de temps total i s'han pogut gestionar els problemes amb agilitat.

Respecte a la metodologia proposada, no s'han fet canvis i s'han realitzat les reunions cada dues setmanes com es va indicar al principi i s'ha incrementat la comunicació a través de correu electrònic i aplicacions de missatgeria instantània, tot i que si bé es cert que durant l'últim més s'ha augmentat la freqüència de les reunions i algun cop s'ha arribat a quedar un o dos cops per setmana per tal de perfilar els últims detalls.

6 Planificació temporal

6.1 Calendari del projecte

El projecte està estimat en una duració aproximada de 4 mesos, de mitjans de setembre de 2017 a mitjans de Gener de 2018. En aquesta estimació en dies entra tota la fase d'autoaprenentatge de la plataforma a utilitzar, investigació i desenvolupament de la solució i preparació final per a la defensa del treball.

Una estimació en hores és: donat que treballarem unes 30h per setmana, tenim sprints de dues setmanes i tenim 9 sprints, això ens dóna una dedicació aproximada de 540h. Com hem dit abans, reservarem un sprint sencer per a desviacions que puguin sorgir, per tant tenim 480h de treball com a tal i 60h de reserva.

6.2 Recursos

Com a recursos per a la realització del projecte, bàsicament en necessitarem dos tipus: recursos humans i recursos materials. Els recursos humans seran el temps del desenvolupador que hi dedicarà i els recursos materials seran, en l'àmbit de hardware i de software, els requisits materials pel correcte desenvolupament del projecte.

6.2.1 Recursos humans

Com a recursos humans, lògicament tenim al desenvolupador, el qual dedicarà unes 30h a la setmana repartides durant cada dia de la setmana. Si bé és cert, mentre duri GEP no es podrà dedicar enterament a la part tècnica sinó que es farà un mix entre part tècnica i part de GEP, però com les hores de GEP també entren dins de la dedicació setmanal estimada, no ens desquadra l'estimació.

6.2.2 Recursos materials

Com a recursos materials utilitzarem molts que ja els tenim disponibles en aquest moment, els d'altres els aconseguirem mitjançant internet donat que seran programari.

- **Hp Zbook Studio G3**: ordinador portàtil on desenvoluparem la gran majoria de la feina, tant tècnica com d'investigació i estudi.
- **Ordinador de sobretaula Intel i5 de quarta generació, 8gb de RAM i Windows 10:** Aquest ordinador serà el que ens farà de *build* agent i test agent, és a dir, serà l'encarregat de donar-nos la integració continua amb el Team Foundation Server.
- ***Unity***: És el programari basic que ens permetrà fer totes les interaccions gràfiques que volem, així com provar i testejar gràficament els algorismes i models estudiats. En el nostre cas utilitzarem la versió Personal, que és gratuïta, ja que ens ofereix tot el que ens pot fer falta pel desenvolupament del treball.
- **Visual Studio 2017 Professional:** Aquest serà el nostre entorn de treball, on podrem desenvolupar. A més a més, des d'aquí controlarem tot el nostre *Backlog, builds, testing*... La versió és de pagament, però la llicencia és la mateixa que utilitzem a la feina, per tant no té un cost directe en nosaltres.
- **Microsoft Team Foundation Server:** Utilitzarem la llicència personal que serveix per petits grups de fins a 5 persones, per tant ja en tenim de sobres. Serveix com a servidor de control de versions de software, a més de controlar tot el tema de *build management* i *testing*, així com el seu feedback.
- **Microsoft Word:** Aquest software l'utilitzarem per a la realització de la documentació, així també per a prendre apunts i notes degudes a les investigacions dels algorismes i models matemàtics que hem d'estudiar.

6.3 Tasques

6.3.1 Descripció de les tasques

Donada la naturalesa del treball i com hem considerat abans, utilitzarem la metodologia àgil *Scrum*, que consisteix en una ràpida adaptació als canvis i aportar valor sempre al final del *sprint*, és a dir, amb *PBI's* que atòmicament ja aporten un valor. Amb *Scrum* no es fan tasques gegants que si estan completades parcialment, no aporten cap valor, sinó que es divideixen en *PBI's* en el que cadascun per si sol, afegeix valor al treball.

En les successives reunions amb els directors del projecte, anirem alimentant el nostre *Backlog* amb diferents *PBI's*. és molt probable que surtin més *PBI's* que hores disponibles en el projecte, així que s'hauran de prioritzar i deixar les menys importants pel final. A més a més, en les discussions amb els directors es pot canviar la prioritat a les tasques per així ser ràpid als canvis.

A mode de tenir un gràfic temporal amb temps i tasques, farem servir un diagrama de Gantt on visualment es podrà veure cada fase del projecte en un context temporal.

6.3.3.1 Gestió del projecte

Aquestes tasques vénen donades per la cronologia de l'assignatura de GEP, per tant tenen una duració aproximada d'un més. Lògicament, no totes les hores que dediquem al treball les dedicarem a fer la documentació de GEP, per tant aquesta tasca estarà encavalcada amb d'altres i es farà paral·lelament.

6.3.3.2 Anàlisi i estudi de publicacions tècniques

Aquesta fase inclou la cerca de material, referències, estudis i publicacions que tractin la temàtica que hem escollit. Inclou cerca a pàgines webs, articles i publicacions d'universitat, llibres, etc. Aquesta tasca serà una tasca recurrent, és a dir, no farem com si fos la metodologia en cascada, d'inicialment estudiar tota la informació que ens interessi, analitzar tots els detalls i sintetitzar-la per a veure si ens serveix, sinó que ho anirem fent de mica en mica mentre vagi avançant el temps.

6.3.3.3 Implementació i verificació de la solució

Igual que l'anterior, aquesta tasca també serà una tasca recurrent, donat que va directament lligada amb la tasca anterior. A mesura que es vagi analitzant els algorismes i models existents, s'aniran implementant i finalment, seran testejats a la nostra solució. Podem dir que tant aquesta tasca com l'anterior, estaran presents a cada *Sprint.*

6.3.3.4 Revisió amb els directors

Aquesta tasca és per a fer una *Sprint Review* amb els directors, és a dir, valorar el que s'ha aconseguit i si hi ha algun *PBI* que no s'ha arribat a assolir, veure els motius i les dificultats associades. Aquest també es moment per parlar i planificar les dues setmanes següents de *l'Sprint.*

6.4 Dependències i precedències de les tasques

Al seguir una metodologia Scrum, per a cada PBI que tinguem tindrem sempre una sèrie de tasques que és repetiran sempre. Això vol dir que tot i que cada PBI tingui un objectiu diferent, un valor concret, tindrem aquestes tasques abstractes i genèriques que seran les que englobaran tota la feina. També mencionar que les tasques, dins de cada PBI, tenen una dependència clara on sempre es la mateixa. En el diagrama de Gantt està mostrat com a fletxes de dependència entre tasques. A continuació les mencionarem també d'una manera escrita, per a cada PBI, quina dependència tenen entre elles:

- Anàlisi i estudi de publicacions tècniques
- Implementació i verificació de la solució
- Revisió amb els directors

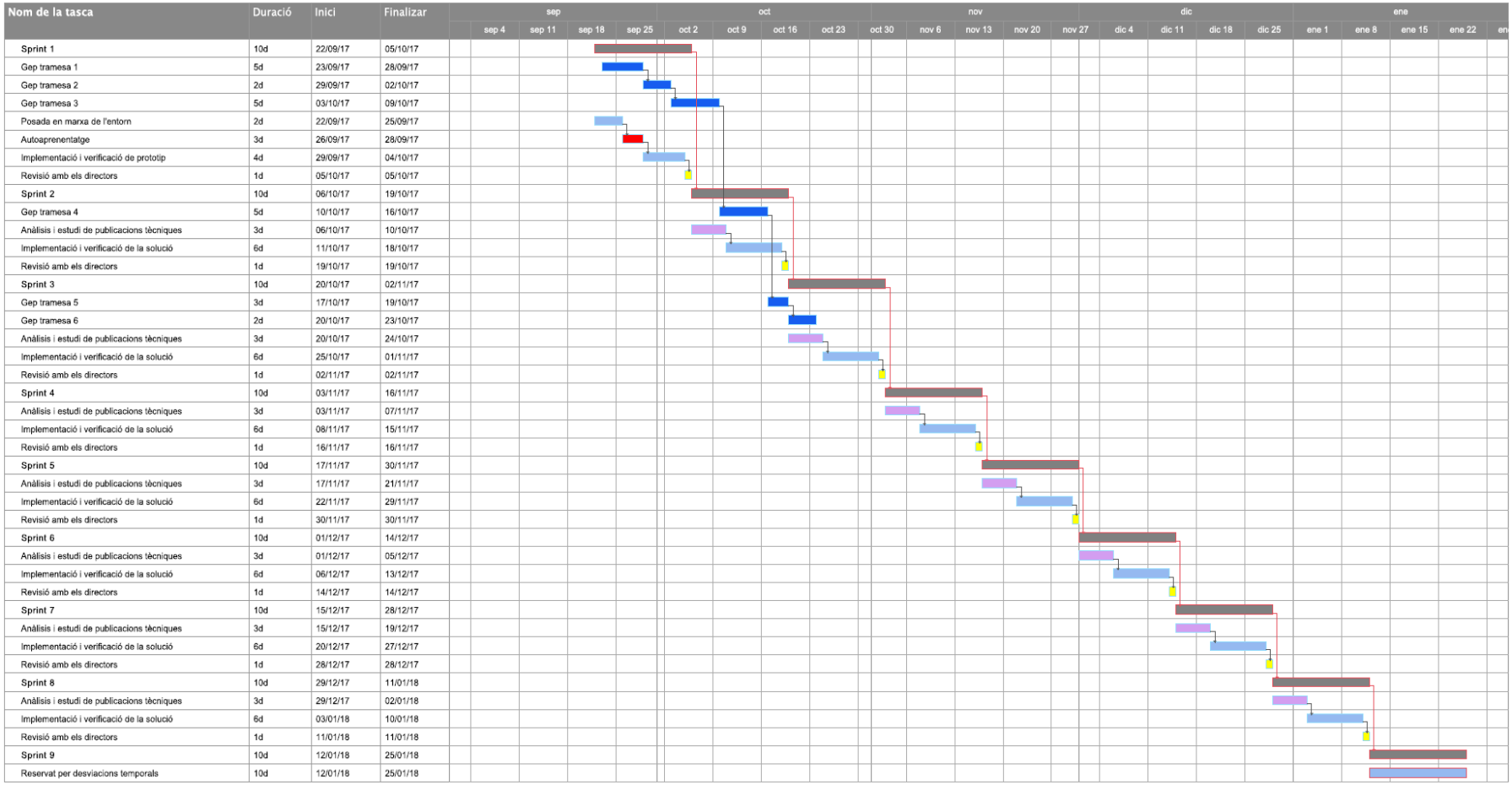
Les tasques de GEP, com hem mencionat abans, no tenen influencia directa amb les tasques més tècniques del projecte, per tant son independents entre si. Òbviament, es segueix la cronologia establerta per les entregues de GEP.

6.5 Estimació de les tasques

Aquestes hores estan extretes del diagrama de Gantt que en la següent pàgina serà mostrat.

Tasca	Hores
GEP	54 h
Anàlisi i estudi de publicacions tècniques	144 h
Implementació i verificació de la solució	324 h
Revisió amb els directors	18 h
Total	540 h

6.6 Diagrama de Gantt



Nom de la tasca	Duració	Inici	Finalizar
Sprint 1	10d	22/09/17	05/10/17
Gep tramesa 1	5d	23/09/17	28/09/17
Gep tramesa 2	2d	29/09/17	02/10/17
Gep tramesa 3	5d	03/10/17	09/10/17
Posada en marxa de l'entorn	2d	22/09/17	25/09/17
Autoaprenentatge	3d	26/09/17	28/09/17
Implementació i verificació de prototip	4d	29/09/17	04/10/17
Revisió amb els directors	1d	05/10/17	05/10/17
Sprint 2	10d	06/10/17	19/10/17
Gep tramesa 4	5d	10/10/17	16/10/17
Anàlisis i estudi de publicacions tècniques	3d	06/10/17	10/10/17
Implementació i verificació de la solució	6d	11/10/17	18/10/17
Revisió amb els directors	1d	19/10/17	19/10/17
Sprint 3	10d	20/10/17	02/11/17
Gep tramesa 5	3d	17/10/17	19/10/17
Gep tramesa 6	2d	20/10/17	23/10/17
Anàlisis i estudi de publicacions tècniques	3d	20/10/17	24/10/17
Implementació i verificació de la solució	6d	25/10/17	01/11/17
Revisió amb els directors	1d	02/11/17	02/11/17
Sprint 4	10d	03/11/17	16/11/17
Anàlisis i estudi de publicacions tècniques	3d	03/11/17	07/11/17
Implementació i verificació de la solució	6d	08/11/17	15/11/17
Revisió amb els directors	1d	16/11/17	16/11/17
Sprint 5	10d	17/11/17	30/11/17
Anàlisis i estudi de publicacions tècniques	3d	17/11/17	21/11/17
Implementació i verificació de la solució	6d	22/11/17	29/11/17
Revisió amb els directors	1d	30/11/17	30/11/17
Sprint 6	10d	01/12/17	14/12/17
Anàlisis i estudi de publicacions tècniques	3d	01/12/17	05/12/17
Implementació i verificació de la solució	6d	06/12/17	13/12/17
Revisió amb els directors	1d	14/12/17	14/12/17
Sprint 7	10d	15/12/17	28/12/17
Anàlisis i estudi de publicacions tècniques	3d	15/12/17	19/12/17
Implementació i verificació de la solució	6d	20/12/17	27/12/17
Revisió amb els directors	1d	28/12/17	28/12/17
Sprint 8	10d	29/12/17	11/01/18
Anàlisis i estudi de publicacions tècniques	3d	29/12/17	02/01/18
Implementació i verificació de la solució	6d	03/01/18	10/01/18
Revisió amb els directors	1d	11/01/18	11/01/18
Sprint 9	10d	12/01/18	25/01/18
Reservat per desviacions temporals	10d	12/01/18	25/01/18

7 Valoració d'alternatives i pla d'acció

7.1 Possibles desviacions i solucions

Donada la naturalesa del projecte, és molt probable que surtin desviacions temporals a causa de problemes que puguin sorgir, males estimacions de les tasques i un llarg etcètera, per això, ens va bé el *Scrum*, ja que tenim la suficient capacitat de reacció per poder reencaminar el projecte ràpidament.

Com hem dit abans, com a mínim, volem disposar de 2 setmanes (1 *sprint*) per poder combatre les desviacions temporals. Considerem que és suficient, ja que amb *Scrum* tenim la suficient agilitat per canviar ràpidament de tasca si alguna és una tasca bloquejant.

Com a desviacions, podem tenir-ne de diferents tipus:

- **Mala planificació:** Per una estimació massa optimista o no considerar totes les tasques d'un *PBI*, ens podem trobar que un PBI estimat per 3 dies se'ns vagi a més i ens desquadri el calendari. Per això és important tenir una visió global del que es vol fer però també una anàlisi profund del que pot implicar.
- **Imprevistos:** Una malaltia, problemes amb la infraestructura... Existeixen moltíssims factors externs que ens poden influir i que difícilment podem tenir control sobre ells. Intentarem que hi hagi el mínim soroll possible.
- **Errors:** Aquí sobretot ens referim en errors de codi, ja sigui perquè no hem entès bé el que se'ns demanava com errors accidentals que podem introduir nosaltres. Per això tindrem tot el tema de *testing* i *building* que ens ajudaran a combatre aquestes situacions. Tot i així, el fet de tenir un *testing* robust també consumeix temps, per tant s'ha de tenir un equilibri.
- **Ineficiència:** Aquest és un perill crític al qual estem exposats. L'eficència en aquest projecte és crítica i es podria optar per una solució la qual en teoria sembla bona però a l'hora de la implementació veure que el rendiment no es l'esperat, reencaminar el projecte en una situació això pot ser complicat, però amb Scrum i les múltiples reunions amb els directors ens poden ajudar.

7.2 Canvis respecte a la planificació inicial

En la realització del projecte, apareixen algunes desviacions eventuals que afecten la duració d'aquest, com poden ser augments en temps de cerca degut a la manca d'informació o augments en temps d'implementació i de proves per culpa dels errors. Tot i això, com s'ha establert una metodologia Scrum, s'han realitzat reunions sovint i d'aquesta manera s'ha evitat un augment de temps total i s'han pogut gestionar els problemes amb poc temps.

Com que el temps que hi ha hagut disponible és molt limitat i les duracions per a cada tasca poden variar pel fet que moltes vegades la incertesa que tenim és alta, si no es pot acabar una tasca al complet per la manca de temps, s'ha assegurat que almenys tenim una versió bàsica i funcional d'aquesta perquè es pugui dur a terme la finalitat del projecte. Com a mínim, tindrem una versió funcional bàsica per a cada tasca indicada.

Ha augmentat el temps esperat de desenvolupament dels algorismes *de Collision Avoidance i Obstacle Avoidance*, però així i tot no representa un impacte al projecte final donat que s'ha anat més ràpid de l'estimat en altres tasques anteriors, com poden ser la posada en marxa de tot l'entorn de TFS gràcies a que ja es tenia bastant experiència en aquest camp.

8 Pressupost i sostenibilitat

8.1 Recursos humans

Donat que és un treball universitari, tota la planificació, desenvolupament i *testing*, recau sobre l'estudiant, en aquest cas l'Alejandro. Tot i això, tindrem en compte les hores dedicades a cada tasca per poder fer un pressupost el més acurat possible.

8.1.1 Team Lead & Scrum Master:

Són dos rols diferents però en molts casos els ocupa la mateixa persona. El *Team Lead* s'encarrega de parlar amb el *Product Owner / Software Project Lead* o similar, i entre ells dos decideixen quines tasques entraran a *l'Sprint*. El *Scrum Master* és l'encarregat de reunir-se amb l'equip *Scrum* i planificar les tasques que entren a *l'Sprint*, és a dir, estimar-la en hores i dissenyar les tasques per a cada PBI. A més a més, és l'encarregat de protegir els interessos de l'equip i de preocupar-se per tots i cadascun dels membres de l'equip, en l'àmbit professional com a laboral.

8.1.2 Software Architect

És l'encarregat de dissenyar la solució en un alt nivell, és a dir, de com hauria de ser la solució. Sol desenvolupar la seva feina amb diagrames de classes, tenint en compte els requeriments i sempre tenint en compte la visió a llarg termini per temes de reutilització, modularitat, eficiència, etc. L'arquitecte és responsable de revisar la qualitat del codi implementada pels desenvolupadors.

8.1.3 Test Manager

Aquesta persona és l'encarregada de dissenyar un *test plan* robust, sempre tenint en compte els requeriments. Això implica que cada requeriment donat, ha d'estar cobert per testos per assegurar que la funcionalitat demanada pels *stakeholders* estan coberts i funcionen correctament. Així i tot, poden existir testos que no cobreixin cap requeriment, són testos de manteniment de la mateixa solució. El *Test Manager*, també és el responsable de la qualitat i robustesa dels testos.

8.1.4 Software Engineer

Com el seu propi nom indica, aquesta persona és l'encarregada de desenvolupar la solució, seguint com a referència els *PBIs* que estan al *backlog*, els quals ja han estat estimats per l'equip, i també seguint les indicacions del *Software Engineer* quant a disseny i del Test Manager pel tema de *testing*.

8.2 Costos

8.2.1 Costos de personal

Donada la dinàmica iterativa que té el projecte gràcies a la metodologia *Scrum*, es basa en *Sprints* de 2 setmanes, de 60 h cadascun. Això implica que les tasques de disseny, anàlisi, implementació i *testing* són comuns a cada *Sprint*. En la següent taula veurem el cost tant temporal com econòmic que tenen els diferents rols al projecte. Tenint en compte el diagrama de Gantt, en total hem calculat que són 54h, per tant aquestes hores les hem inclòs com a una tasca separada a les tasques típiques d'un *Sprint*.

La component temporal l'hem tret del diagrama de Gantt, analitzant les tasques de cada *Sprint* i assignant a cada tasca les hores estimades.

La component econòmica l'hem calculat tenint en compte els diferents rols que hem considerat. Per a fer-ho hem consultat diferents webs de referència en aquesta temàtica [24] [25] [26].

	Rol	Hores	€/h	Total
Documentació de GEP	*Software Architect*	56 h	23.45€	1.313€
Estimació dels PBIs	*Team Lead & Scrum Master*	18 h	26	468 €
Anàlisis i recerca d'informació	*Software Architect*	108 h	23.45€	2.532€
Disseny de la solució	*Software Architect*	54 h	23.45€	1.266€
Implementació	*Software Engineer*	270 h	18	4.860 €
Testing	*Test Manager*	54 h	19	1.026 €
		540 h		11.465 €

Taula 1: *Pressupost de recursos humans*

8.2.2 Costos de material

En aquesta part hem considerat els costos de material necessari per a desenvolupar el projecte. Hem tingut en compte tant el hardware com el software necessari.

Producte	Cost	Vida útil	Amortització (4 mesos)
Hp Zbook Studio G3	3200 €	4 anys	333 €
Ordinador de sobretaula	950 €	5 anys	63 €
Visual Studio 2017 Professional	500 €	3 anys	55 €
Microsoft Team Foundation Server	0	-	0 €
Microsoft Word	139 €	4 anys	11 €
			463 €

Taula 2: *Costos de material del projecte*

8.2.3 Costos generals

En aquesta part hem considerat els costos de generals no però que estan involucrats d'alguna forma en el projecte. Poden ser la llum, el transport, internet, etc.

Producte	Cost	Període	Amortització (4 mesos)
Electricitat	0.2kWh * 0.17kW	540 h	18 €
Internet	30 € / mes	4 mesos	120 €
Transport	2 T10 de 2 zones	20 Viatges	40 €
			178€

Taula 3: *Costos generals del projecte*

8.2.4 Costos imprevistos

Tenint en compte que pràcticament tot el treball es desenvolupa amb un ordinador, sembla sensat pensar que els imprevistos poden venir d'aquí. Els problemes més típics que podem tenir són avaries pel que fa a el hardware que ens impossibilites treballar, per això es disposa d'un ordinador personal MacBook Pro 13" amb pantalla retina com a còpia de seguretat, per tant, en cas de tenir problemes amb un es podria utilitzar l'altre.

És reservarà un 10% del subtotal del pressupost com a mecanisme de control per si existeix algun imprevist que es indisposa l'ús de l'altre màquina, ja sigui algun problema tècnic o com haver de dedicar més hores de feina

8.2.5 Cost total del projecte

La suma total ve donada pels costos de personal, material, despeses generals i el fons reservat per a contingències.

Tipus de recurs	Cost
Personal	11.466,1 €
Material	463,81 €
Despeses generals	178,36 €
Subtotal	12.108,27 €
Contingència (10%)	1.210,83 €
Total	13.319.10 €

***Taula 4**: Costos totals del projecte*

8.3 Control de gestió

Com a mètodes de control per a la part de recursos humans, gràcies a *Scrum* i a què tenim un *backlog* al *TFS* on anem apuntant les hores de cada PBI i cada tasca, podem tenir calculada a la perfecció la dedicació en hores que dediquem a cada tasca, PBI i *Sprint*, per tant tenint això actualitzat, podem veure les desviacions que puguin haver-hi al projecte, ja sigui tant a l'alça com en mancança d'hores de dedicació.

Com a mètode de control de recursos materials, al només tractar-se d'un ordinador, els costos romandran fixes durant tot el projecte. No són necessàries llicències de software de l'apart mencionades ni comprar cap tipus de material extra, per tant el cost serà invariant. L'únic que podem tenir en compte és si sorgeix alguna avaria, problema pel qual ja hem discutit una solució anteriorment.

També donat que tenim un *Sprint* de marge, el *Sprint* 9, tenim 60 h de marge de maniobra que estan estimades i pressupostades, per tant, en cas que tot vagi bé, aquestes hores es dedicaran a ampliar tot el que es pugui el projecte.

9 Sostenibilitat i compromís social

9.1 Matriu de sostenibilitat

Després d'analitzar el projecte i veure com podem respondre a les preguntes sobre aquesta competència, la matriu ens quedaria d'aquesta manera amb 63/90 punts. No és gaire alta donada a què en l'aspecte social no canvia gaire l'existència o no existència d'aquest projecte. Els majors impactes es tenen en l'àmbit personal, en el consum del disseny, la petjada ecològica i el pla de viabilitat. En canvi, de riscos en tenim pocs, nomes un cert risc ambiental donat al petit increment en temps que comporta un augment en el consum energètic.

Sostenibilitat	PPP	Vida Útil	Riscos
Ambiental	Consum del disseny	Petjada ecològica	Riscos ambientals
	8/10	17/20	-2
Econòmica	Factura	Pla de viabilitat	Riscos econòmics
	7/10	18/20	0
Social	Impacte personal	Impacte Social	Riscos Socials
	9/10	6/20	0
Rang sostenibilitat	24	41	-2
	63/90		

Taula 5: *Matriu de sostenibilitat*

9.2 Impacte econòmic

El cost econòmic ve donat pel cost total que hem calculat anteriorment. La major part del cost de diners ve donat per gestos de recursos humans, és a dir, hores, i donat que és un estudiant el que realitza aquesta feina i no un equip complet de desenvolupadors, el cost és molt menor. S'han utilitzat també ordinadors personals i software pel qual ja es tenia la llicència, ja sigui per l'empresa on treballa l'estudiant o llicències gratuïtes per convenis entre la UPC i Microsoft.

El projecte és *Open Source,* per tan qualsevol empresa o desenvolupador de jocs el podria incorporar, provar, estendre sense cap cost, per això el podria utilitzar qualsevol qui vulgui sense cap comprimís. La idea principal del projecte és millorar funcionalitats que venen per defecte a *Unity*, per tant amb un cost de 0, qualsevol ho podria implantar al seu videojoc o projecte.

El projecte no requereix d'actualitzacions periòdiques amb noves funcionalitats, si mes no, no significa que en un futur no es pugi millorar algun aspecte del projecte, com millores de rendiment, amb el qual això s'hauria de tenir en compte perquè implicaria un cost addicional.

Es complicat reduir costos i recursos al projecte donat que el temps de dedicació ha sigut el just i imprescindible, per tant per aquest cantó, no es podrien reduir costos.

Quant al projecte, s'han establert mecanismes de control de desviacions, tant de gestos com d'hores, com el 10% de previsió de fons que tenim per s'han de fer més hores o el *Sprint* extra que el tenim per a cobrir possibles desviacions horàries. Donada la investigació en l'estat de l'art, es tenen molt clar els objectius i com arribar-hi.

9.3 Impacte social

Aquest projecte es troba dins del món del software, més concretament dintre del món dels videojocs. El projecte com a tal no representa un benefici social, ja que es resumeix tot al món virtual dels videojocs per tant, no millora en cap aspecte res sobre la vida de les persones.

Es un projecte fet amb codi obert, multi plataforma i es penjarà a la web perquè tothom el pugui utilitzar sense cap compromís, per tant és totalment gratuït i es pot estendre la seva funcionalitat.

En l'àmbit personal ha aportat molts coneixements en temes de realització de videojocs i comportament realista i a més s'ha pogut treballar i descobrir tecnologies que estan a l'ordre del dia.

9.4 Impacte ambiental

Durant el desenvolupament d'aquest projecte, no s'ha utilitzat res més que ordinadors personals, per aquest motiu, l'únic impacte ambiental directe és el consum elèctric, pel qual ja hem vist que és molt petit.

El ser un projecte informàtic, un dels avantatges que te és l'escassa utilització de paper, tinta d'impressora, bolígrafs, etc. Amb això es redueix el consum i els residus que es poden generar. A més a més, es disposa d'una pissarra a la paret per si s'han de fer càlculs o apuntar qualsevol cosa.

Donat que és de codi obert i totalment gratuït, es pot reutilitzar o ampliar si es vol, per tant d'aquesta manera es poden estalviar recursos en temps que a la vegada repercuteix al medi ambient en dedicar menys esforç i energia en reinventar la roda ja que aquest projecte es pot utilitzar com a base d'altres mes complexos o per fer algun tipus de prova.

10 Unity

10.1 Introducció a Unity

Unity és un motor de videojocs que intenta facilitar l'accés al món del desenvolupament de videojocs. La primera versió del motor es va publicar l'any 2005 però no ha estat fins als últims anys que ha guanyat molta popularitat, sobretot en l'entorn dels desenvolupadors de videojocs independents. Ofereix una gran quantitat d'eines, suport per a 2D i 3D, i permet exportar per a un gran nombre de plataformes.

L'estructura de *Unity* és la següent: La zona o zones on es treballa i es pot visualitzar tot el que es realitza s'anomenen escenes, i en ella es troben tots els objectes que inclourem en l'entorn, com les estructures o els personatges. Tot objecte dins d'una escena s'anomena ***GameObject***, i aquest està format per components. Les **components** són les propietats de l'objecte, com per exemple la ***Transform***, que ens indica la posició, rotació i escala en *local space* de l'objecte, o els scripts que serviran per descriure el comportament dels objectes.

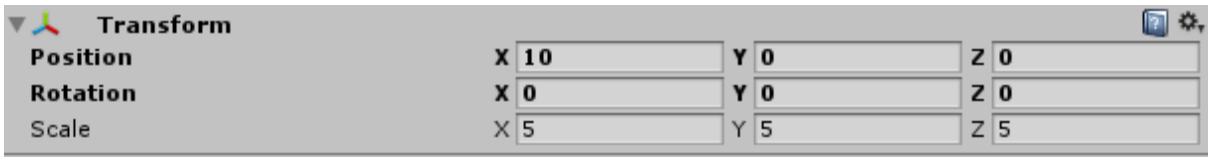


Figura 4: Transform de un GameObject

Unity també proporciona tota una bateria de models simples com poden ser els cubs, esferes, cilindres, plans, etc., per a poder crear el nostre petit mon virtual d'una manera ràpida i fàcil. Si es volen utilitzar models més complexos com ara un cotxe, una persona o un arbre, s'han d'utilitzar models específics que no venen per defecte al *Core* de *Unity.*

Podem trobar molts models (gratuïts i de pagament) a la botiga [27] de *Unity* on tenim diversos filtres per a realitzar cerques de models, *plugins*, extensions del programa, etc., que ens puguin interessar. No cal reinventar la roda quan segur que ja existeix i ho podem utilitzar.

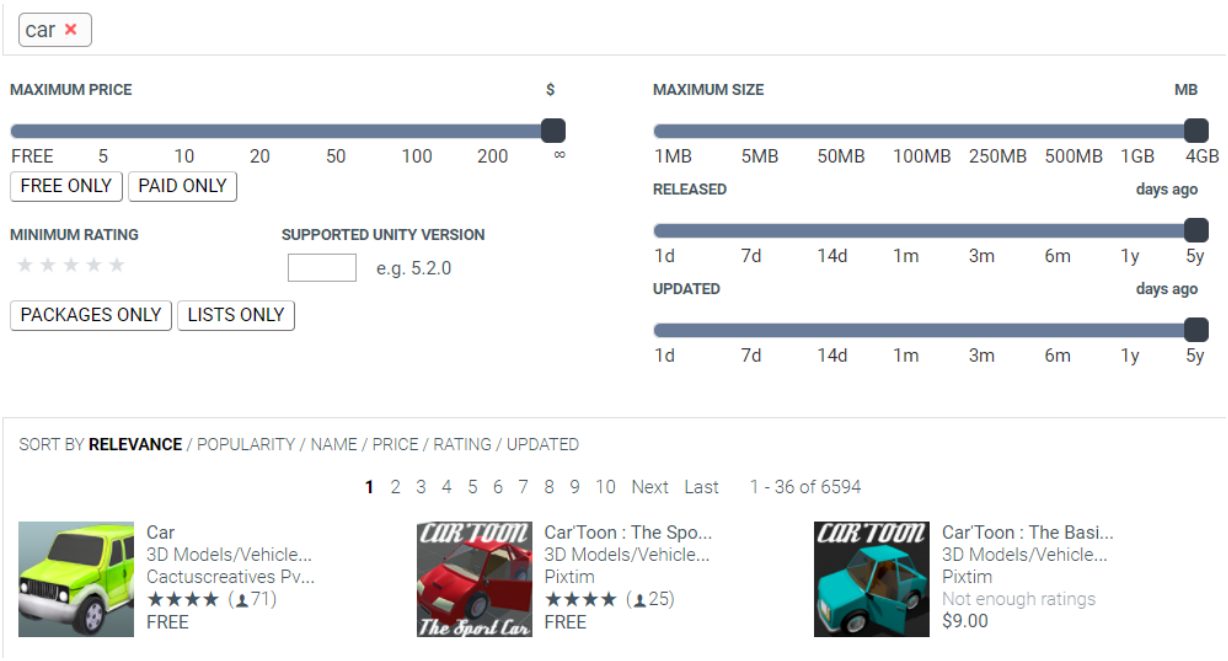


Figura 5: Exemple de cerca a la Unity Store

Així i tot, es posen desenvolupar models amb eines externes de modelatge, tant lliures com de pagament. Tenim exemples com **Blender** [28], **Maya** [29], **3D Max** [30]... Estudiant les diferents funcionalitats de cadascun podem saber quin seria el més adequat.

Tot i que queda fora de l'abast del treball i es treballarà per sobre, *Unity* també conté moltes més funcionalitats a les abans esmentades, com poden ser les animacions, sons, il·luminacions, efectes visuals, etc. Com hem dit abans, és un entorn complet en el qual poder desenvolupar un joc.

Una part molt important en *Unity* i també en aquest treball són els ***Colliders***. Els *Colliders* defineixen la forma d'un objecte per a propòsits de col·lisions físiques. Són invisibles i aproximen la forma de l'objecte. Aquest component és imprescindible per la simulació de trajectòries, un dels temes més importants en la simulació de multituds.

Per exemple, si tenim un cub, existeix un *Collider* cub i llavors el model i el *Collider* coincideixen en la forma, però si tenim un cotxe, no existeix cap *Collider* que tingui aquella forma exactament. El que es fa és una composició de *Colliders* "primitius" i s'aproxima, d'una manera acurada, la forma del model i dotar de col·lisions físiques a parts complexes d'un model.

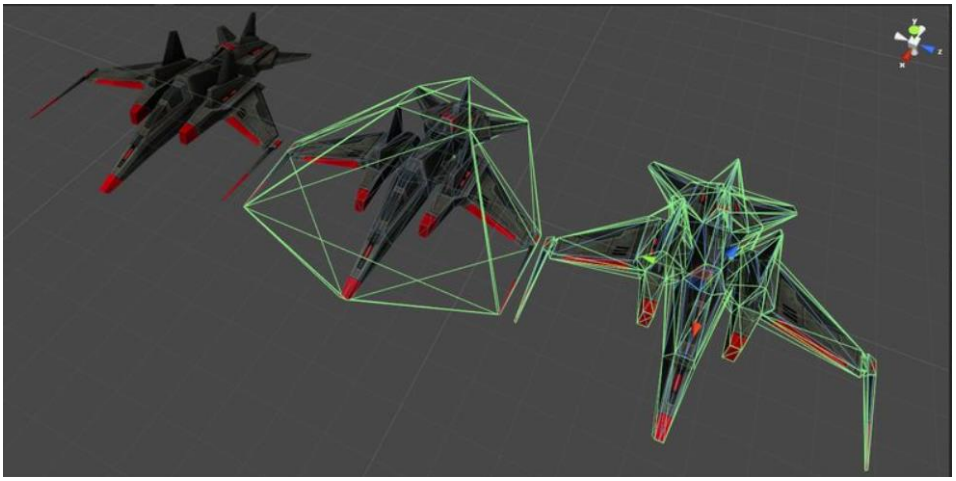


Figura 6: Exemple de model sense *collider*, *collider* simple i *collider* complex

10.2 Tecnologia utilitzada

Unity permet la creació *de* ***scripts*** per a crear, modificar o substituir el comportament del joc, ja que proporciona una *API* [31] molt completa. Com a exemple del que es pot fer amb els *scripts* seria fer que un objecte es mogui i segueixi el punter del ratolí, tenir un generador d'obstacles on es pugui definir el numero i la zona on es vol i els generi aleatòriament... És en aquest punt on es desenvolupa la gran part del treball.

Unity permet la utilització de diferents tecnologies per a la creació dels scripts. Nosaltres hem escollit C# que utilitza el *framework* de ***Microsoft .Net Framweork*** [32]. *Unity* permet utilitzar dues versions de .Net, la versió 3.5 i la versió 4.6. Per defecte utilitza la 3.5 però és una versió antiga i la 4.6 ofereix moltes millorar de rendiment i del propi *framework* en si [33]. El "problema" és que la versió 4.6 és experimental en *Unity*, per tant ens avisa que podem tenir un comportament inestable si la utilitzem.

Nosaltres ho hem fet i el rendiment i comportament ha sigut excel·lent. En les pròximes versions de *Unity* pesarà a ser una versió estable i marcarà com a obsoleta l'antiga versió. Així mateix, amb la versió 4.6 de .Net, podem utilitzar la versió 6 de C#, la qual implementa moltes millores respecte de la 4 que és la que utilitza .Net 3.5 [34] [35].

En el propi codi de C#, importem biblioteques de C# com del mateix *Unity*, que són la manera que ens permet accedir a la seva API, com per exemple accedir a la posició (accedint a la seva *Transform*) x ,y, z d'un *GameObject* i saber en quina posició x, y i z està o modificar la forma del seu *Collider* en temps d'execució.

Unity implementa, mitjançant crides a la seva API, la majoria de característiques que hem hagut d'utilitzar, com per exemple, canviar la posició d'un objecte, la física de les col·lisions, gravetat, llums, creació d'agents, etc.

El que hem implementat són tots els comportaments com moure un objecte d'un punt a un altre, esquivar objectes, tan fixes com en moviment, fugir davant un objectiu... Tots aquests comportaments seran descrits en profunditat en els següents apartats.

11 Desenvolupament del projecte

11.1 Prerequisits

En aquest apartat resumirem els prerequisits que hem hagut de complir per a començar el desenvolupament del projecte.

El que s'ha fet com a començament (a part de l'obvi de la instal·lació de tot el software com Unity, Visual Studio, etc.) ha sigut començar per un senzill tutorial [36] de Unity.

En ell s'introdueixen conceptes bàsics de Unity, com la **Càmera** (el punt des d'on es veu l'escena), els *Game Objects* amb els seus *transform* que contenen el seu sistema de coordenades, els ***Rigidbodys*** que implementen tota la física de l'objecte (com pot ser la gravetat, el pes, reacció a les col·lisions ...) i també els *colliders*, que com hem vist anteriorment, són una de les parts més importants d'aquest projecte.

El tutorial realitzat consisteix en un petit escenari quadrat, on hi ha uns rombes que giren i el joc consisteix en controlar una bola que es mou en el teclat i agafar els rombes que proporcionen una determinada puntuació.

Aquí podrem veure algunes captures de pantalla del tutorial realitzat.

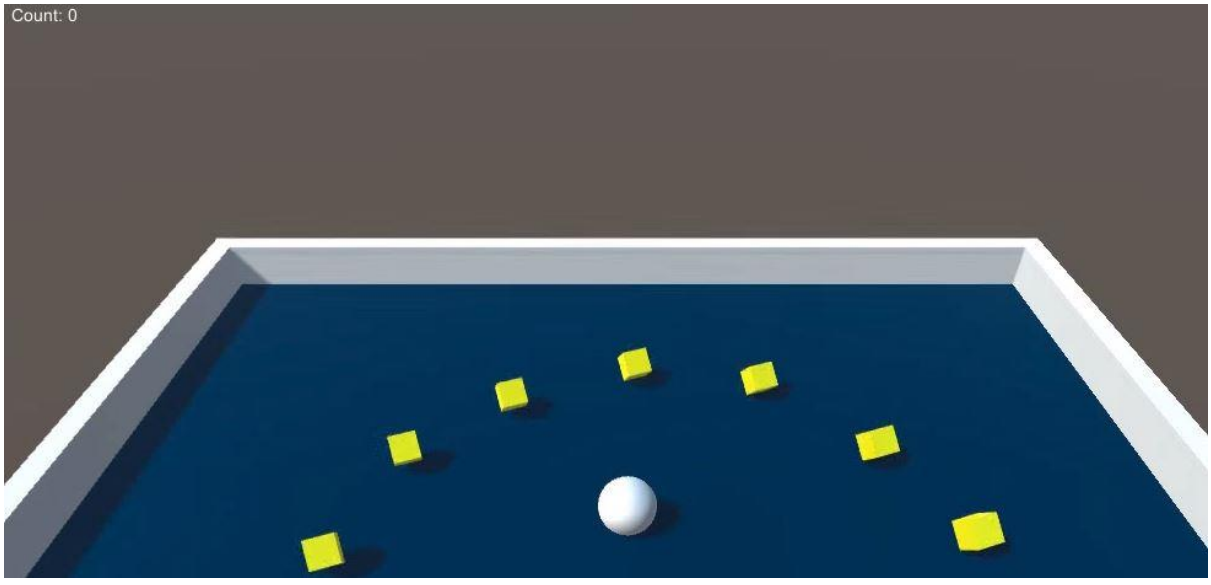


Figura 7: Tutorial de Unity (Inici)

Si ens fixem a la part superior esquerra, ens indica la puntuació total aconseguida fins al moment.

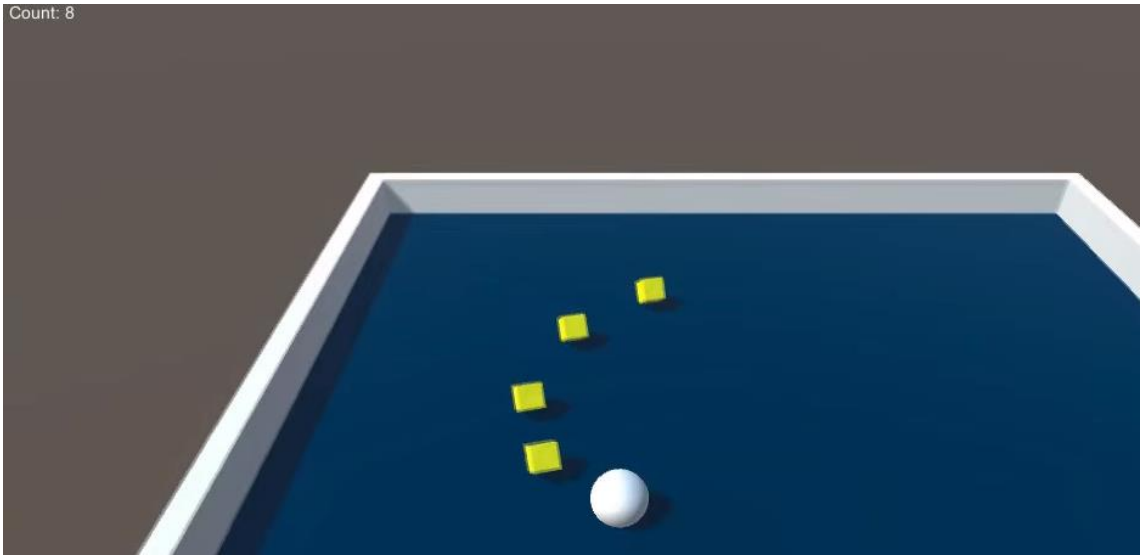


Figura 8: Tutorial Unity (Jugant)

Aquí podem veure com a mesura que anem passant per sobre dels objectes de color groc, desapareixen i ens proporcionen punts a la cantonada esquerra de la pantalla.

L'objectiu d'aquest tutorial és, donada la manca d'expertesa en *Unity*, hem cregut convenient dedicar unes hores en fer-nos amb tota la interfície de Unity i desenvolupar un petit joc amb ajuda del tutorial oficial de *Unity*. També s'ha volgut invertir temps a entendre els conceptes bàsics de *Unity* i com integrar comportaments programats amb C# utilitzant com a editor el Visual Studio 2017.

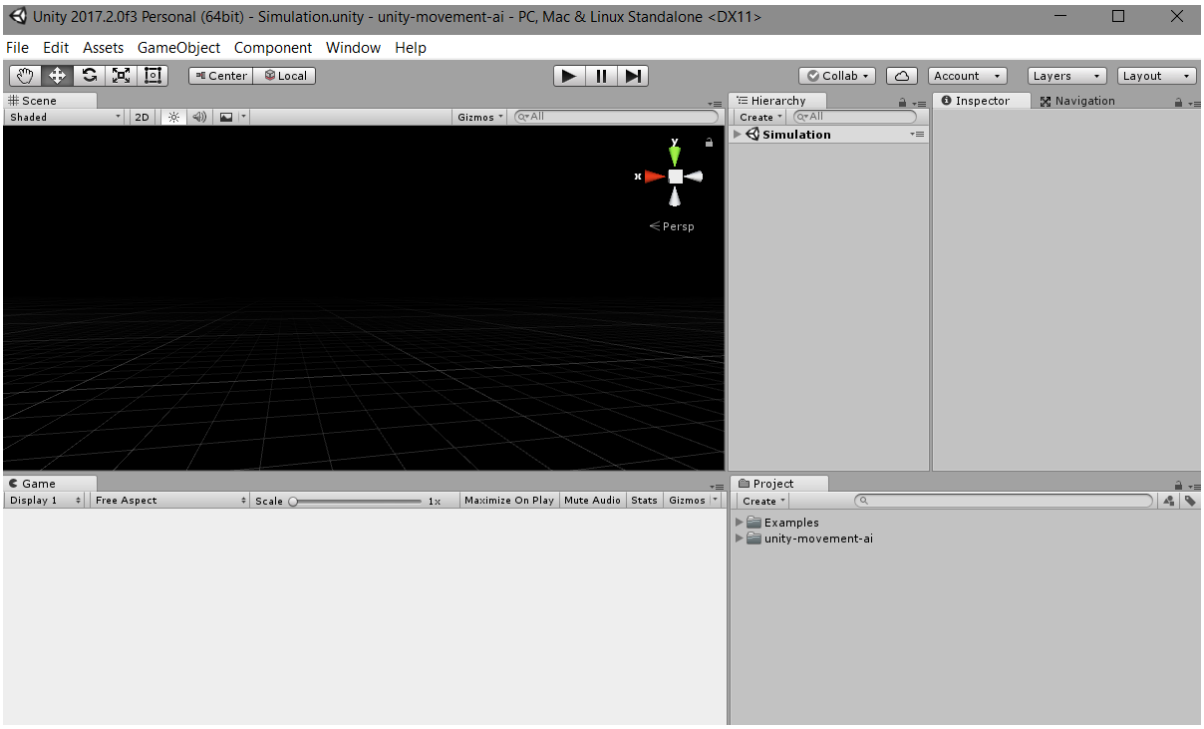


Figura 9: Interfície de Unity

Ha sigut també el punt de partida per a crear el repositori de codi amb el *Team Foundation Server*, fer algunes proves de configuració i integració amb Visual Studio, pujar alguns canvis del codi i veure com s'executa una *build definition.* Al definir els *steps* que ha d'executar podem veure a la següent figura que hi ha un que fa la còpia del codi font d'un lloc a un altre, un altre que fa la compilació del codi i finalment, l'últim pas que publica els resultats.

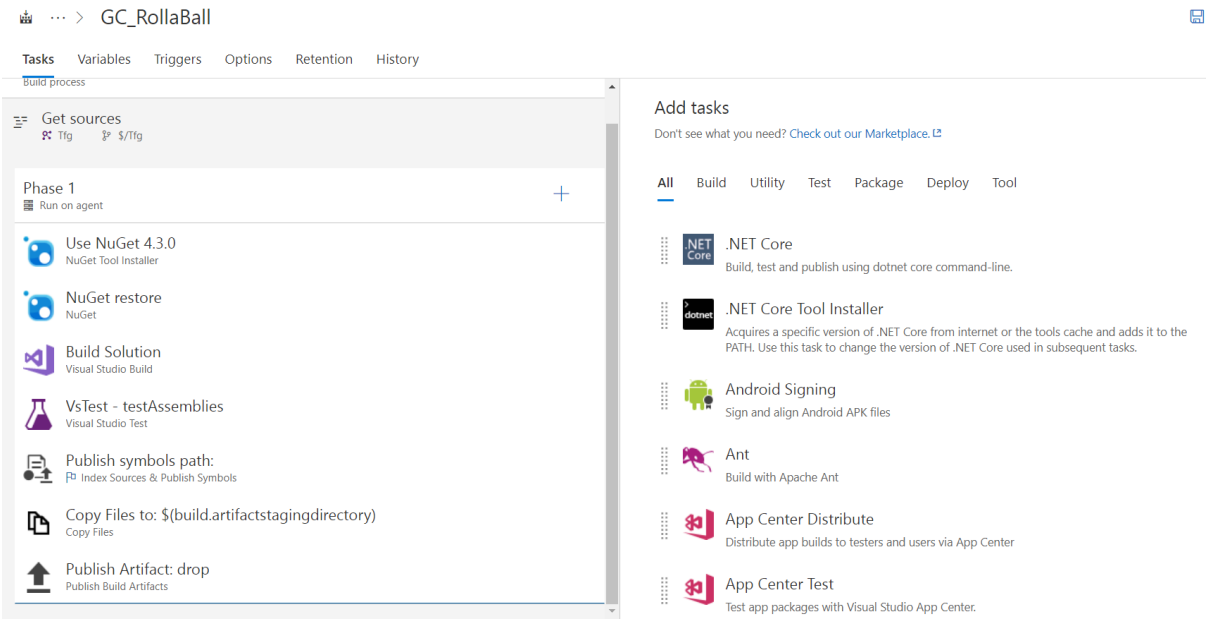


Figura 10: Build Definition utilitzada en el tutorial

Com podem veure a la figura anterior, podem afegir més tasques a la nostra *build definition* per si ens interesses fer algun altre procés, com per exemple fer el *deployment* de l'aplicació per IOS o Android, publicar el resultat de l'execució en alguna pàgina web, pujar-la a algun *cloud*...

11.2 Primers passos

Un cop entesos els primers conceptes bàsics de Unity i tenint ja tota la infraestructura provada i llesta, ens hem posat mans a l'obra amb la implementació del projecte en si.

El que primer hem fet és tota la creació del nostre entorn de proves: escenari, agents, càmera, il·luminacions...

11.2.1 Creació de l'entorn

El primer pas que hem fet és la creació del nostre entorn, és a dir, en quin món estaran els nostres agents.

Serà en principi, un món molt senzill. No hi haurà terra, és a dir, els agents i obstacles estaran volant. El motiu és que és indiferent que hi hagi o no terra perquè al final serem nosaltres qui decidirem en quins eixos de coordenades es poden moure, per tant en el nostre cas, es mouran en el pla XY i tindrem la Z bloquejada a 0 per tal que no hi hagi cap efecte de profunditat. Podria ser també els eixos XZ amb la Y bloquejada a 0, però per temes de configuració de la càmera de Unity, és més senzill tenir-ho d'aquesta manera, però es podria canviar si es volgués sense cap problema. Al final l'únic que ens interessa és un món en dues dimensions.

En temes d'il·luminació i càmera deixarem els valors per defecte. La càmera només l'enfocarem perquè estigui en una distància determinada enfocant a la zona on estaran tots els agents.

11.2.2 Creació dels agents

Com a primera aproximació a un agent, en tenim un que és de molt simple: consisteix en una esfera que té un con davant, com si fos el cap d'un ocell.

Aquesta primera aproximació és senzilla de crear amb un programa com *Blender* que ens permet crear l'objecte d'una manera molt ràpida, simplement afegint un con a una esfera.

Hem decidit aquesta forma primerament perquè, com hem dit abans, és una manera molt ràpida de tenir un agent perquè es crea ràpidament i segon i més important, el con de la figura apuntarà a la direcció on es va movent l'agent, per tant d'una manera molt visual es pot veure la direcció que té aquell agent en qualsevol moment donat.

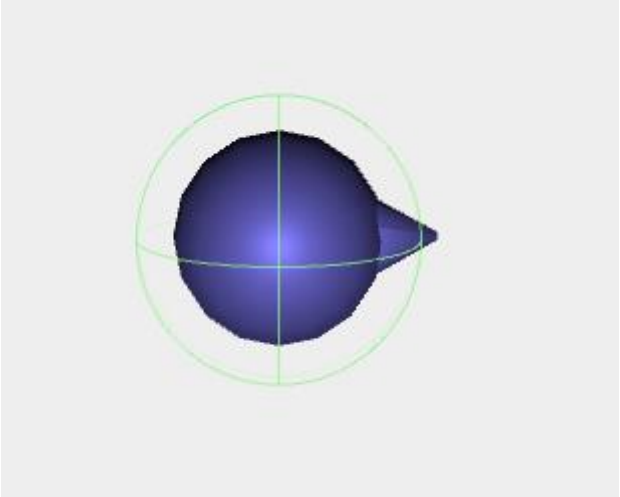


Figura 11: Forma d'un agent

Com podem observar també, el nostre agent implementa un *collider* esfèric (la part verda de la Figura 11) que és una mica més gran que l'objecte en si per tal de donar un "espai vital" una mica més ampli del qual tindria si fos purament ajustat.

Implementa també un *rigidbody* que com hem pogut veure anteriorment, proporciona a l'objecte qualitats físiques. El nostre *rigidbody* té desactivada la gravetat, si l'activéssim, al no tenir terra, cauria fins a "l'infinit". La resta de valors romandran per defecte, ja que no els utilitzarem per res.

La manera de crear els agents, és a dir, posar-los al mapa és molt senzilla. Primerament hem de crear un *GameObject* buit. Després hem d'anar assignant-li característiques utilitzant el botó de *Add Component* com podem veure a la seguent figura.

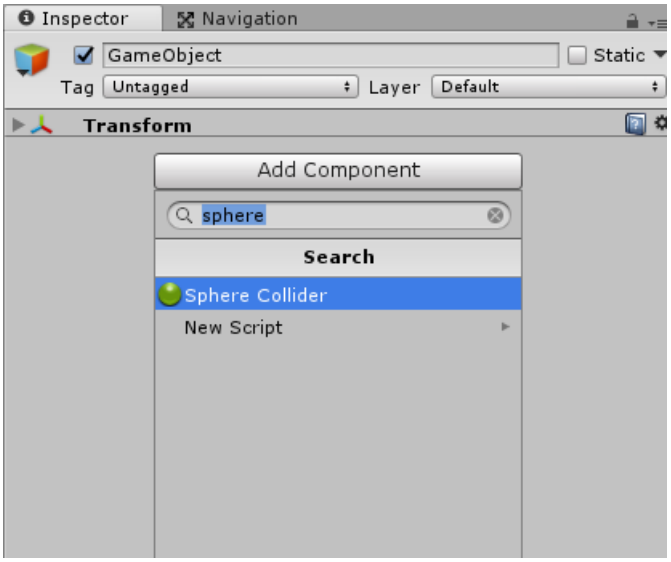


Figura 12: Manera d'afegir comportaments a un GameObject

Figure 1

Quan s'obre el quadre de text per afegir les components, ens apareix també un quadre per fer cerques, com per exemple un *Sphere Collider* o qualsevol altre cosa. Podem afegir també comportaments ja creats escribint-los al camp de cerca també, però la manera més fàcil és agafant el script directament i arrosegant-lo fins al GameObject.

Un cop tenim creat el nostre agent amb totes les components i comportaments que volguem, una bona manera de diferenciar un d'un altre es canviant-li el nom i posar-li algun tipus de nom que ens ajudi a identificar que és, ja que si en tenim agents amb comportaments diferents, ens pot costar diferenciar un agent d'un altre.

La manera que utilitzem nosaltres per diferenciar els agents es classificar-los per tipus. Per exemple. Tenim SeekUnit, PursuitUnit, HideUnit... més endavant veurem en detall aquests comportaments i que volen dir cadascún però, d'aquesta manera tenim identificats i classificats els agents pel comportament que implementen.

Un cop tenim els agents creats, els podem assignar la característica de que siguin una plantilla, és a dir, podrem crear clons d'agents d'una manera senzilla i ràpida. Unity anomena les plantilles d'objectes com a *Prefabs*. Una altre gran característica dels *Prefabs* és que es poden modificar totes les instancies de *GameObject* d'aquell *Prefab* simplement modificant els atributs de la plantilla. Això és molt util per canviar les característiques de tots els agents, com la velocitat o d'altres propietats dels comportaments.

11.2.3 Implementació de la lògica d'un agent

Tal com tenim ara mateix el nostre projecte no fa res. Tenim només un agent que no implementa cap lògica, per tant el nostre personatge no fa res.

Definirem a un agent com:

- **MaxVelocity:** la màxima velocitat que pot arribar l'agent.
- **MaxAcceleration:** la màxima acceleració que pot tenir l'agent.
- **TargetRadius:** Radi de l'objectiu pel qual l'agent creu que està suficientment a prop i identifica que ha arribat.
- **SlowRadius:** Distancia fins a l'objectiu en el qual l'agent comença a frenar.
- **Massa:** la massa de l'agent.
- **MaxForce:** la força màxima que te l'agent a l'hora de moure's. Això es veurà més endavant amb els comportaments.
- **TurnSpeed:** Velocitat de gir.

Amb aquesta informació, tots els nostres agents heretaran d'aquesta classe i implementaran aquestes funcionalitats, ja que són funcionalitats comunes. Després si un agent o un altre té alguna peculiaritat com veurem més endavant, implementaran el seu propi comportament, però qualsevol agent ha de complir les definicions anteriors.

A la següent figura veurem com es veu un script que està adjuntat a un *Game Object,* en aquest cas, el nostre agent.

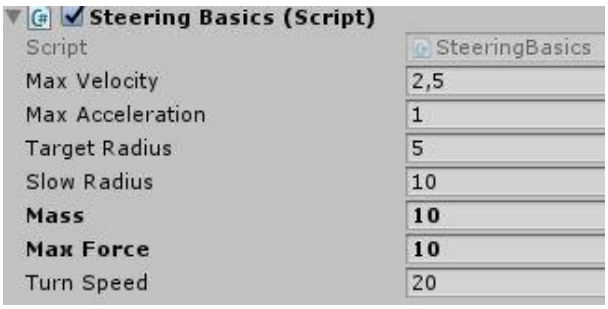


Figura 13: Script de definició de l'agent

No apareix en la definició de dalt perquè no és un valor modificable com els anteriors, però tots els agents contindran un *Rigidbody* que inclourà la posició actual en la qual es troben i la seva velocitat.

Totes aquestes característiques tenen un petit factor aleatori perquè cada personatge sigui diferent i que sigui més realista.

11.3 Comportaments

Hem començat implementant els comportaments descrits al paper de Craig Reynolds anomenat **Steering Behaviors For Autonomous Characters** [19], la nostra base de tot el projecte.

11.3.1 Introducció als comportaments

L'article de Reynolds de Steering Behaviors ens explica com ajudar als agents a moure's d'una manera realista, mitjançant l'ús fe forces simples que es combinen per a produir una navegació realista al voltant dels agents.

Una característica d'aquest article és que els comportaments no es basen en estratègies complexes com la planificació de rutes o càlculs globals, sinó que utilitza informació local, com les forces d'un agent i els seus veïns. Això fa que sigui fàcil d'entendre i d'implementat, però tot i així, es poden aconseguir patrons de moviments molt complexos com podrem veure més endavant.

La implementació de totes les forces descrites a l'article es poden aconseguir utilitzant vectors matemàtics. Donat que aquestes forces influiran en la velocitat i la posició del personatge, també seran representades utilitzant vectors.

El vector de velocitat indicarà cap a on es mou l'agent, és a dir, és un vector que indica una direcció, mentre que la magnitud (o longitud) del vector indicarà quant es mou a cada *frame* de *Unity*, que per defecte, la diferència entre dos *frames* són 0,02 segons. Com major sigui la longitud del vector de la velocitat, més ràpid es mou el personatge. La velocitat del vector de velocitat es pot truncar per a garantir que no es passa de la velocitat màxima que hem indicat als paràmetres de l'agent anteriorment.

Si no utilitzem res més, el personatge cada cop que es mogui cap a l'objectiu serà d'una manera abrupta i poc realista. Imaginem que l'agent va recte i canviem l'objectiu i està a un cantó seu, el gir serà instantani. La idea de l'article de Reynolds és influir en el moviment del personatge mitjançant l'addició de forces de direcció d'aquí el títol de comportaments de direcció (*Steering Behaviors)*. Depenent d'aquestes forces, l'agent es mourà en una o en una altra direcció.

11.3.2 Comportament de Seek

El primer comportament descrit a l'article de Reynolds és el comportament de *Seek*. Aquest comportament implica moure's cap a un objectiu.

En aquest comportament, l'addició de forces de direcció al personatge en cada *frame* fa que la seva velocitat tingui un ajustament suau, evitant així canvis sobtats de ruta. Si l'objectiu es mou, el personatge canviarà gradualment el seu vector de velocitat, tractant així d'arribar de nou a la seva nova ubicació.

El comportament de *Seek* implica dues forces: velocitat actual i velocitat desitjada com podem veure a la següent figura:

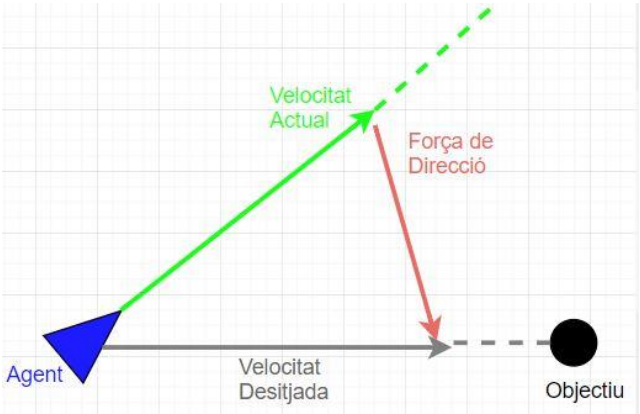


Figura 14: Comportament de Seek

La velocitat desitjada és una força que guia al personatge fins al seu objectiu utilitzant el camí més curt possible, que és una línia recta entre ells. La força de direcció *(Steering Force*) és el resultat de la velocitat desitjada restada de la velocitat actual i és aquesta la que empeny a l'agent fins a l'objectiu.

Després de calcular la força de direcció, s'ha de sumar a la velocitat de l'agent. Aquesta suma de forces a cada *frame*, farà que el personatge abandoni suaument la seva antiga ruta recta i es dirigeixi fins a l'objectiu, descrivint una ruta de cerca (*Seek Path*) d'una manera suau com podem observar en la següent figura:

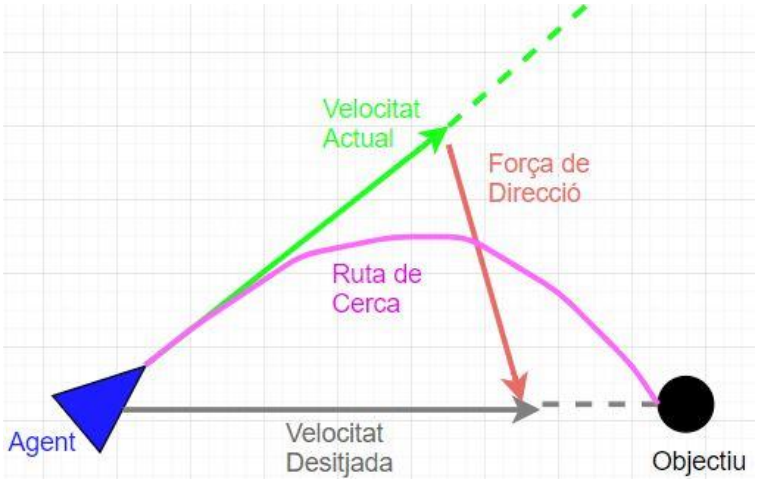


Figura 15: Comportament de Seek amb la ruta de cerca

Un cop calculada la força de direcció, trunquem el vector per assegurar-nos que no excedim el màxim de força permesa anteriorment definit a les característiques de l'agent. També dividim el vector entre la massa de l'agent, aconseguint així un efecte més real que simula el pes. Finalment sumem del vector de la velocitat amb la força de direcció i la trunquem amb la velocitat màxima. El resultat d'aquesta operació se suma a la posició actual i ja tenim el nostre moviment.

Cada cop que l'objectiu es canvia de posició, la velocitat desitjada de l'agent canvia en conseqüència. El vector de velocitat es pren un temps per canviar i comença a apuntar a l'objectiu novament. El resultat és una transició de moviment suau.

Aquest és un dels comportaments més importants del nostre projecte, molts dels següents comportaments, com podrem veure, vindran derivats o directament utilitzaran aquest comportament.

11.3.3 Comportament de Flee

Aquesta implementació implementa el comportament de fugir d'un objectiu fent que s'allunyi d'un punt.

A l'igual que abans, el comportament de *Flee* implica dues forces: velocitat actual i velocitat desitjada com podem veure a la següent figura:

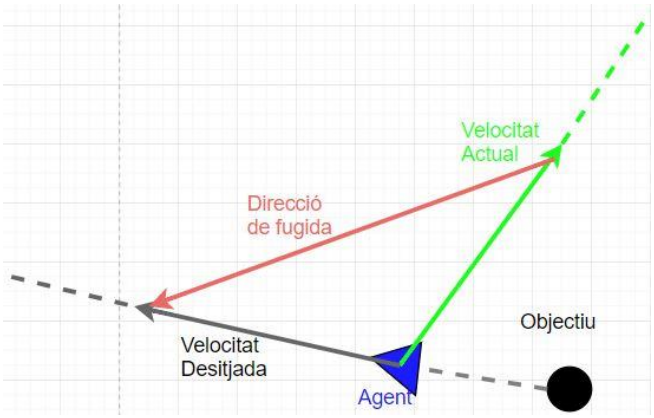


Figura 16: Comportament de Flee

La nova velocitat desitjada és calculada restant la posició actual de l'agent de la posició de l'objectiu, la qual produeix un vector que va de l'objectiu fins al personatge. En aquest cas, la velocitat desitjada representa la ruta més senzilla per a fugir, empenyent a l'agent cap a la direcció del vector de velocitat desitjada.

Com podem observar, aquest comportament és com el de *Seek* però negant el vector de la velocitat desitjada, així doncs, podem concloure un comportament és el negat de l'altre.

Així doncs, podem veure a la següent figura com seria la ruta de fugida que adoptaria el nostre agent:

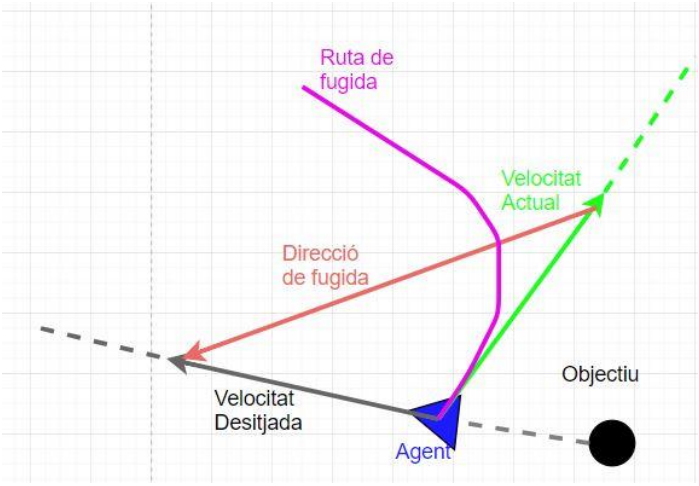


Figura 17: Comportament de Flee amb la ruta de fugida

La suma de totes les forces com hem comentat al comportament de *Seek*, tenint en compte la massa, la força màxima i la velocitat, romandran igual en tots els comportaments, ja que això és genèric per tots els comportaments. Com també abans, obtenim un efecte suau i progressiu aplicant aquest comportament.

Per millorar aquest comportament se li podria aplicar que només fugis de l'objectiu si està a una certa distància, ja que en aquest cas, fugiria d'ell estigui a la posició que estigui i que pari quan es trobi a una certa distància també, perquè en aquest cas fuig indefinidament.

11.3.4 Comportament de Arribe

Aquest comportament implica una millora substancial en el comportament de *Seek*, perquè tal com el tenim ara mateix, mai frena quan arriba a l'objectiu, per tant es queda orbitant al seu voltant indefinidament.

El comportament d'arribada evita que el personatge es mogui a través de l'objectiu, fa que el personatge minori la seva velocitat a mesura que s'apropa al destí, detenint-se així quan arriba finalment a l'objectiu.

El comportament està composat de dues fases. La primera fase és quan el personatge està lluny de l'objectiu i funciona exactament de la mateixa manera que hem descrit al comportament de *Seek*. La segona fase és quan el personatge està prop de l'objectiu, dins de l'àrea de frenada (un cercle centrat a la posició de l'objectiu) com podem veure a la següent figura:

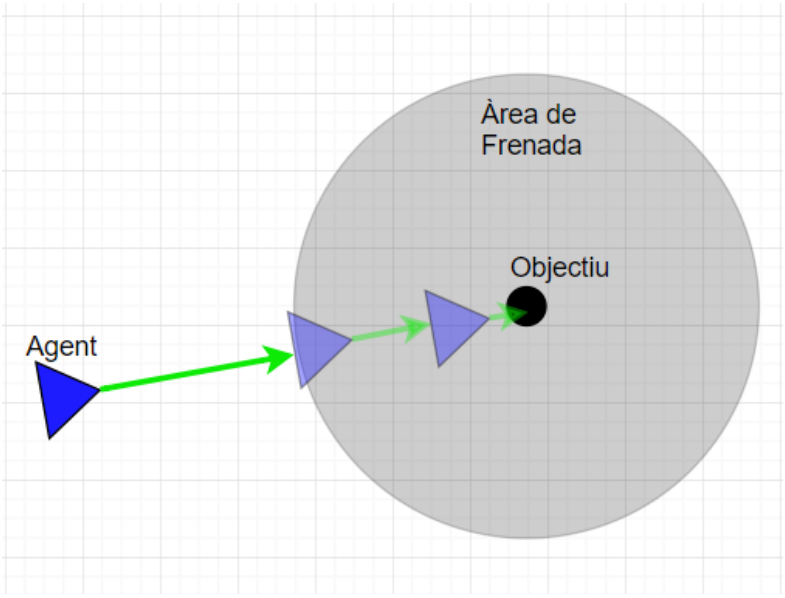


Figura 18: Comportament de Arribe

El procés és gradual i es calcula en funció del radi de l'àrea de desacceleració i la distancia entre el personatge i l'objectiu.

Com hem dit abans, tenim dues fases:

- Si la distancia és major que el radi de frenada (el qual hem definit abans a les característiques d'un agent), el comportament de *Seek* és normal perquè l'agent està lluny de l'objectiu.
- Si la distància és menor al radi de frenada, vol dir que l'agent ha entrat a l'àrea de frenada i la velocitat ha de ser reduïda. Utilitzant la distancia fins a l'objectiu i el radi de frenada, tenim que els valors aniran d'1 (quan la distancia sigui igual al radi de frenada, és a dir, es trobarà al límit exterior del radi de l'objectiu) a 0, que voldrà dir que ja hi és a sobre. Aquesta variació lineal farà que la velocitat disminueixi suaument.

11.3.5 Comportament de Wander

Aquest comportament produeix una navegació improvisada, fa que l'agent es mogui aleatòriament pel món. Aquest comportament té moltes variants com podrem veure a continuació.

La manera més simple i més ràpida d'implementar aquest comportament és que l'objectiu canviï cada X segons de localització fent així que l'agent es mogui d'un lloc a un altre perseguint l'objectiu. Això fa que mai es pari de moure perquè l'objectiu sempre està canviant de posició.

Tot i que sembla un bon enfocament del problema, el resultat final no és del tot el que desitjaríem. A vegades el personatge inverteix completament la seva ruta perquè l'objectiu s'ha situat just darrere seu, per tant tot i tenir els comportaments suavitzats com hem vist abans, realitza canvis de sentit que no són tan realistes com ens agradaria.

A l'article de Reynolds això ho té cobert i ell fa una implementació diferent. La idea principal és produir un petit nombre de moviments aleatoris i aplicar al vector actual del personatge, que en el nostre cas és la velocitat, a cada *frame* del joc. Com el vector de velocitat defineix a on es dirigeix l'agent, qualsevol interferència canviarà la ruta actual.

La utilització de petits desplaçaments en cada *frame* evita que el personatge canviï abruptament de ruta. Si el personatge es mou cap amunt i gira a la dreta, en el pròxim *frame* de joc seguirà pujant i girant a la dreta, però en un angle lleugerament diferent.

Aquest enfocament també es pot implementar de diferents maneres. Reynolds tal i com ho descriu al seu article, és utilitzant un petit cercle al davant del personatge, utilitzant això per calcular totes les forces involucrades com podem veure en la següent figura:

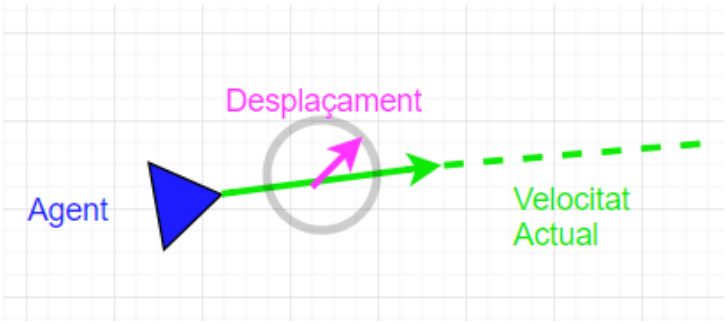


Figura 19: Comportament de Wander

La força de desplaçament té origen en el centre del cercle i està restringida pel radi del cercle. A més gran sigui el centre, més força tindrà la força de desplaçament per empènyer al nostre agent a cada *frame.* Això es coneix com a *Wander Force*.

El primer requeriment que ens cal és calcular la posició del centre del cercle. Com que estarà situat davant del nostre agent, podem utilitzar el vector de velocitat com a guia.

Es pot fer simplement clonant el vector de velocitat el qual ens apunta a la direcció que ens interessa, es normalitza i es multiplica per un escalar que ens indica la a la que volem que estigui el centre del vector respecte al nostre agent.

El següent requeriment és obtenir la força de desplaçament que és la responsable d'interferir amb la nostra direcció. Com que ens interessa qualsevol distorsió, aquest vector pot apuntar a qualsevol direcció.

La força de desplaçament és creada i escalada pel radi del cercle. Com hem descrit anteriorment, a més gran sigui aquest radi, més forta serà la força de desplaçament

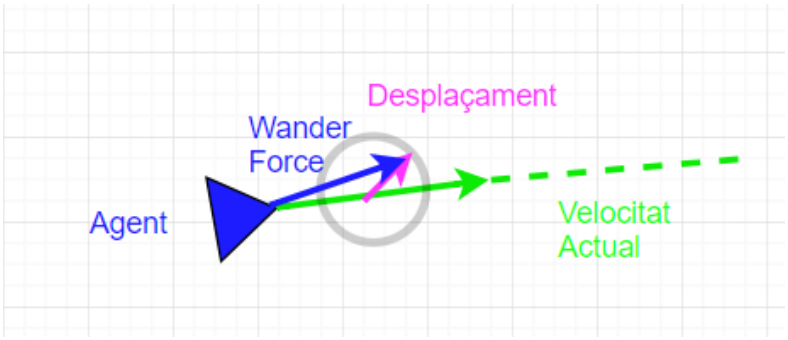


Figura 20: Comportament Wander amb la Wander Force

La *Wander Force* treballarà exactament com les forces de *Seek* o *Flee*, empenyerà l'agent en la seva direcció. D'una manera similar als comportaments de *Seek* i *Flee* on la força és calculada basada en la posició de l'objectiu, aquí la força es calcula amb el punt aleatori dins de la circumferència.

11.3.6 Comportament de Follow

En aquest comportament veurem que un agent persegueix a un altre agent. La idea principal és seguir a l'objectiu i estar el mes a prop possible d'ell.

Aquest comportament és molt simple i intuïtiu d'implementar doncs només és passar a cada *frame* la posició de l'objectiu i aplicar el comportament de *Seek* descrit anteriorment.

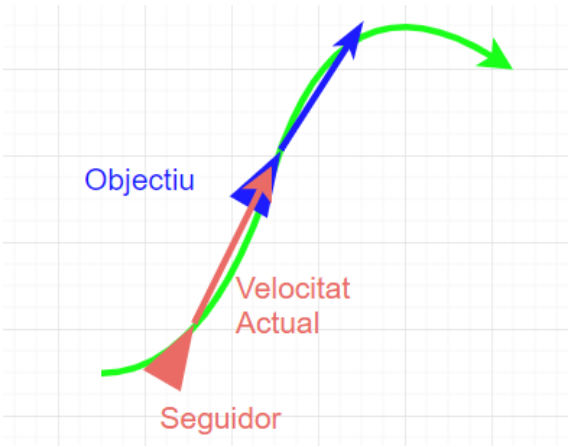


Figura 21: Comportament de Follow

11.3.7 Comportament de Pursuit

Aquest comportament implementa el procés de seguir un altre agent amb l'objectiu d'atrapar-lo, aquesta és la principal diferencia amb el comportament de *Follow* que simplement el segueix. El comportament de *Follow* com hem vist abans, simplement repeteix els moviments de l'objectiu i com a conseqüència, sempre anirà a la seva cua.

Quan s'aplica el comportament de *Pursuit*, en algunes ocasions el perseguidor simplement segueix a l'objectiu però la característica principal és que s'anticipa a on l'objectiu estarà un futur pròxim. Si podem estimar on estarà l'objectiu en els pròxims *frames*, és possible ajustar la ruta per escurçar camí i evitar rutes innecessàries.

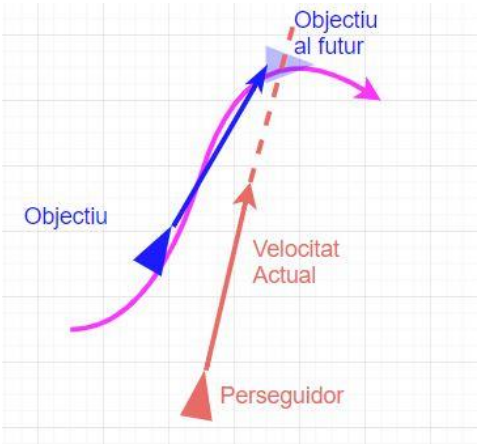


Figura 22: Comportament de Pursuit

Aquest comportament és molt semblant al *Seek*, la diferència és que el perseguidor no persegueix a l'objectiu en si, sinó a la seva posició futura. La implementació més simple d'aquest comportament és agafar la posició de l'objectiu i sumar-li el producte de la seva velocitat per un factor T que serà la seva predicció.

Amb aquest càlcul obtenim una estimació de la posició futura i aquesta informació la hi passem al *Seek* perquè es desplaci cap allà.

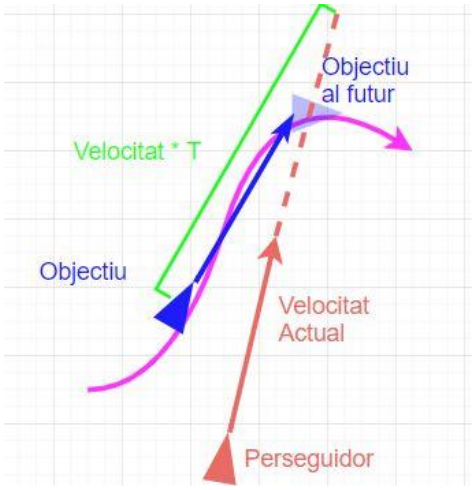


Figura 23: Comportament de Follow utilitzant un factor T constant

El problema d'aquesta aproximació és que aquest factor T és un valor constant. A més a prop estigui el perseguidor de l'objectiu, pitjor serà la predicció que seguirà predient T *frames* més enllà.

Hi ha una solució senzilla per aquest problema i millora en gran mesura la perícia del perseguidor. Necessitem un valor T que no sigui constant, això ho podem aconseguir amb la distància entre els dos personatges dividida entre la màxima velocitat que l'objectiu pot aconseguir.

A mesura que la distancia entre els dos sigui més gran, més gran serà aquest factor T per tant el perseguidor buscarà un punt futur molt per sobre de l'objectiu. En canvi, quan la distància sigui curta, T tindrà poc pes, trobant així un punt futur pròxim a l'objectiu com podem veure en les dues figures següents:

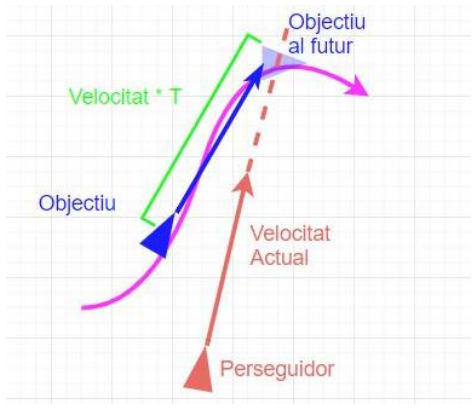


Figura 24: Comportament de Follow amb T dinàmica (T gran)

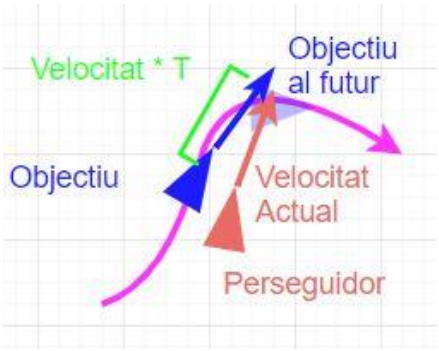


Figura 25: Comportament de Follow amb T dinàmica (T petita)

11.3.8 Comportament de Evade

Aquest comportament és l'oposat al comportament de *Follow*. En comptes d'anar a la posició futura de l'objectiu, fugirà d'ella. A l'igual que la comparació del *Seek* amb el *Pursuit*, aquest és comparable amb el *Flee,* ja que no fuig d'un objectiu en concret sinó de la seva posició futura.

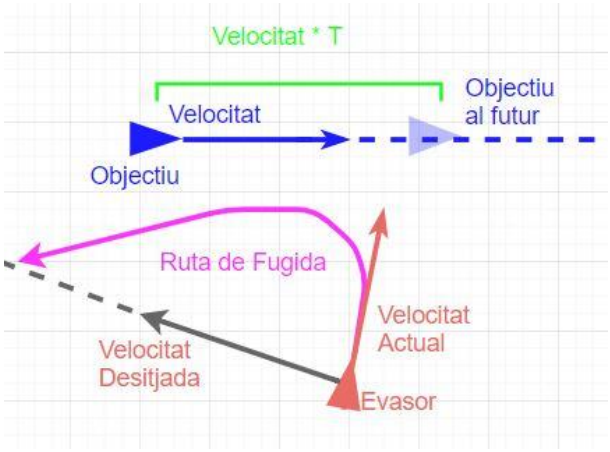


Figura 26: Comportament de Evade

El codi del *Evader* és idèntic al de *Pursuit*. L'única diferència és que en comptes de passar la posició futura al *Seek* com al *Follow*, li passem la posició futura al *Seek*.

11.3.9 Comportament de Hide

Aquest comportament no està descrit per l'article de Reynolds i és una aportació pròpia. Ens ha semblat interessant implementar-lo, ja que la idea és molt simple i és un comportament curiós.

Imaginem que tenim al nostre món cilindres que fan d'obstacle, com si fossin columnes disposades aleatòriament, després tenim un agent que implementa el comportament de Wander (l'anomenarem **W** a partir d'ara a fi de tenir una nomenclatura mes simple), que com sabem va navegant aleatòriament pel mapa i després tenim un agent que és el que implementa el comportament de Hide (**H** a partir d'ara) que es vol amagar d'aquest agent.

Aquest comportament es divideix en dues fases. La primera fase és quan l'agent **H** encara no té cap obstacle entre ell i l'agent **W.** En aquest cas, l'agent **H**, busca ràpidament quin és l'obstacle més proper i se'n va cap a ell.

Figura 27: Comportament de Hide fase 1

La segona fase del comportament és quan l'agent **H** s'amaga de l'altre agent. La idea principal aquí és que ens cal la posició de l'agent **W** i de l'obstacle i també quin marge volem deixar amb l'obstacle.

En pseudocodi seria:

```
float distAway = obstacle.radius + obstacleOffset
Vector3 dir = (obstacle.position – wanderingAgent.position).Normalize()
return obstacle.position + dir * distAway
```

Això ens retorna la posició que ha d'anar l'agent H per amagar-se de l'altre agent. El punt retornat és el punt més distant amb l'altre agent, per tant estarà amagat de tal manera que l'altre agent no el podria veure.

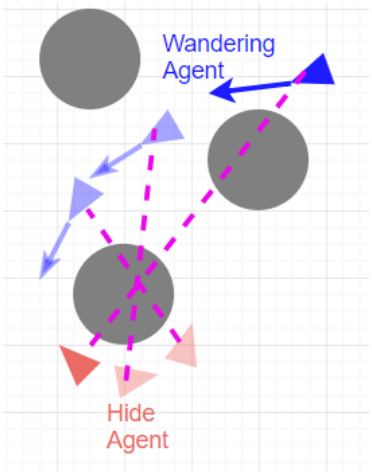


Figura 28: Comportament de Hide fase 2

Com podem apreciar a la figura anterior, l'espai entre l'agent H i l'obstacle al qual s'està amagant és l'espai (que hem anomenat *obstacleOffset* en el pseudocodi anterior) entre l'agent i l'obstacle de la figura anterior. Si incrementem aquest offset, la separació amb l'obstacle serà major, o menor si decrementem aquest valor.

11.3.10 Flocking

Aquest comportament es podria descriure d'una manera fàcil si diguéssim que seria com viatjar en grup, com un grup d'animals a la sabana que van a l'uníson, o un banc de peixos que es manté agrupat però ordenat. Això no és un comportament per si sol com podríem pensar, sinó que és la suma de 3 comportaments: cohesió, separació i alineament.

Aquests comportaments són molt típics al món animal i nosaltres els podem simular. Reynolds va pensar originalment en aquests tres, però un d'ells, l'alineament, *Unity* el té resolt ja per defecte, ja que l'alineació en si és simplement una línia de codi,[1] no s'ha de fer cap càlcul, per tant ja tenim un problema resolt.

11.3.10.1 Comportament de Cohesion

Aquest comportament causa que els agents es vagin agrupant entre si formant grups d'agents més nombrosos i que es mouen com a grup i no com a individus.

La idea principal és calcular el centre de massa o centre de la gravetat. Cada agent té una àrea d'influència parametritzable que retorna els veïns propers. L'agent comprova els seus veïns i es fa un sumatori de totes les posicions dels veïns. Finalment aquest resultat és dividit pel nombre de veïns de l'agent. Aquest resultat és el centre de gravetat.

Això ens retornarà un punt a l'espai que indica una coordenada. Aquesta és el punt on l'agent s'ha de desplaçar, per tant li passem al comportament *Arribe* (que a la vegada implementa el *Seek*) i l'agent es desplaça amb suavitat fins al centre de gravetat.

Tot això es fa per cada agent a cada *frame*, per tant com podem observar pot ser costós si l'àrea dels veïns és molt gran. Però això pot no ser un problema, ja que tampoc cal una àrea d'influència gran, perquè cada agent en tindrà la seva, per tant amb àrees relativament petites es poden cobrir amplies zones del món.

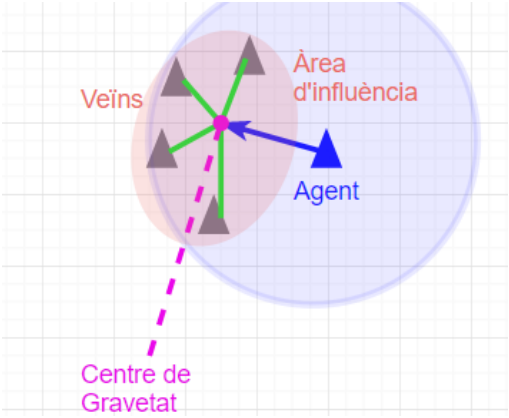


Figura 29: Comportament de Cohesion

[1] https://docs.unity3d.com/ScriptReference/Quaternion.LookRotation.html

11.3.10.2 Comportament de Separation

Aquest comportament com és de suposar, manté una separació entre els agents evitant així que col·lisionin per proximitat.

Bé, això no és del tot cert. Si pensem sobre la frase evitar que col·lisionin per proximitat. Imaginem-nos un cotxe, les rodes giren i pot evitar obstacles sempre que, i aquí entren molts factors, el cotxe no perdi adherència, no vagi excessivament ràpid i molts d'altres factors.

En aquest cas, pot passar que la força de repulsió entre els agents no sigui suficient i puguin xocar. Si ho pensem bé, això és un comportament del més natural. El que queda artificial és que sigui com sigui, mai puguin col·lisionar, ja sigui perquè van massa ràpids i apliquem una força de repulsió desmesurada o perquè ignorem la massa que tenen evitant així el seu comportament natural.

Amb això volem dir que s'ha de trobar un equilibri entre esquivar objectes que siguin esquivables i xocar quan no hi hagi més remei que és el comportament natural.

En comportaments de més endavant veurem tècniques més avançades per donar un efecte de realisme en el tema d'esquivar, tant agents com obstacles, i que tot i així resulten naturals i suaus.

Tornant al comportament de separació, la implementació és molt similar al comportament *Cohesion*. En aquest cas, volem separar-nos dels veïns que la seva distància amb l'agent sigui menor a cert valor definit per l'usuari per tant hem de calcular una força de separació.

Aquí és on entra la força que hem comentat abans, ja que un dels valors de la força final és una força introduïda per l'usuari. Si és massa gran, sempre es separarà però el resultat no serà natural, en canvi, si és massa petita, col·lisionaran amb massa facilitat.

En pseudocodi seria:

```
Vector3 ForçaSep = La força abans mencionada introduïda per l'usuari.
float MaxSepDist = Distància mínima que volem estar separats d'un veï.
Vector3 direction = agent.position – vei.position
float dist = Distància entre l'agent i el veí = direction.magnitude
Vector3 acceleration = On acumularem totes les forces de separació

float strength = ForçaSep * (MaxSepDist - dist) / (MaxSepDist – agent.radi – veï.radi)
acceleration += direction.Normalize() * strength
```

Això ho fem per a cada veí massa proper a l'agent i el resultat ens donarà el punt on ens hem de desplaçar. Com també amb el *Cohesion*, aquí enviem el punt donat al comportament *Arribe* i ens desplaçarà suaument fins a aquella posició.

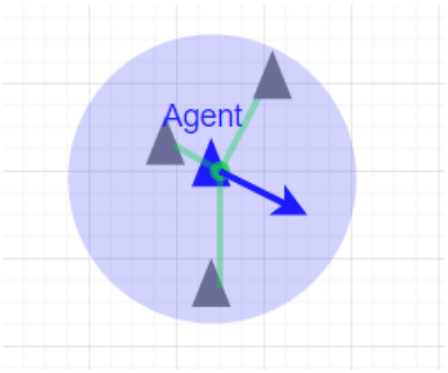


Figura 30: Comportament de Separation

11.3.11 Path Following

Aquest comportament implementa la capacitat de què un agent segueixi un camí predefinit compost de punts al nostre món. Aquest comportament també pot ser implementat d'un gran nombre de maneres. L'enfocament que utilitza Reynolds per solucionar aquest problema és la utilització d'un camí compost per línies i els agents la segueixen estrictament, com si fossin raïls de tren.

Aquesta situació és massa estricte perquè no deixa pràcticament marge de maniobra als agents, ja que s'han de lligar estrictament a la línia i això comporta que el comportament no sigui del tot realista. Una manera de simplificar el problema i guanyar molt realista sigui que la línia no sigui com un raïl, sinó com una referència a la qual l'agent ha d'anar seguint-la el màxim possible, però sense tenir la restricció de seguir-la sempre com si fos un tren. Amb això el que guanyem és tenir més llibertat de moviment.

Un camí pot ser definit com un conjunt de punts (vèrtexs) connectats per línies (arestes). Encara que les corbes poden ser utilitzades per a descriure un camí, un camí de punts i línies és molt més simple de fer i produeix el mateix resultat gràcies al comportament que hem vist al *Seek*, que suavitza els moviments per anar d'un objectiu a un altre.

La manera més simple d'aconseguir aquest comportament és passar al comportament de *Seek* quin és l'objectiu al qual ha d'arribar en tot moment, i un cop arribat, canviar d'objectiu al següent vèrtex de la llista de vèrtex que representa el camí.

El simplisme d'aquesta implementació si mes no, comporta una sèrie de problemes com que l'agent, tot i que no segueix estrictament el camí pels "raïls", sí que ha de passar per a sobre de l'objectiu. Com a conseqüència d'això podem tenir comportaments inesperats com que l'agent es quedi donant voltes una estona sobre l'objectiu perquè no ha pogut arribar exactament al punt que vol.

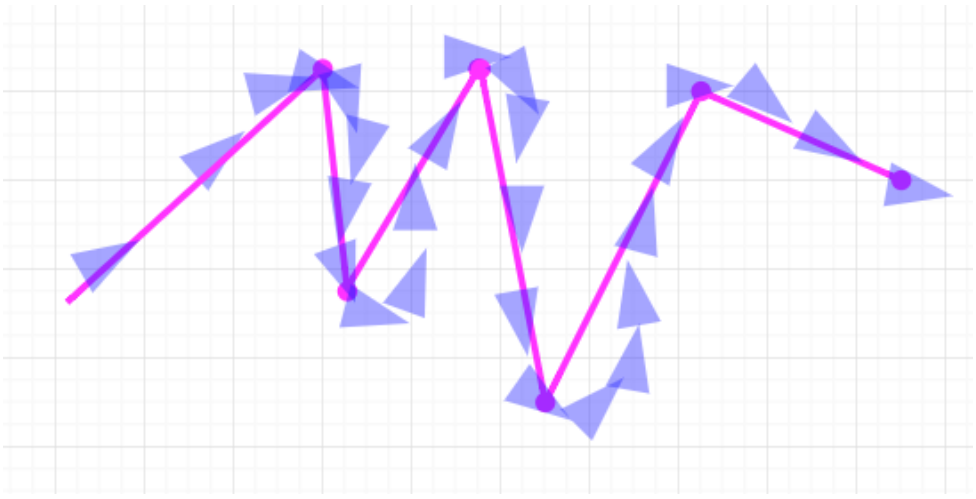


Figura 31: Path Following passant per sobre dels punts

Una manera senzilla de solucionar aquest problema és eliminant la restricció que l'agent hagi de passar exactament pel punt, si no que cada punt tingui una àrea d'influència i que a l'agent només li calgui passar per algun punt de l'àrea del punt per poder canviar d'objectiu.

Aquest comportament és similar al comportament de *Arribe* que hem mencionat abans, ja que també treballa amb el radi de l'objectiu però no és del tot igual. Si la distancia entre l'agent i el punt és menor o igual al radi de l'objectiu, podem dir que s'ha arribat al punt, com a conseqüència d'això aconseguim un moviment amb molta més llibertat que abans.

A més gran sigui el radi, més amplia serà la ruta i major serà la distància que mantindran els personatges de l'objectiu. El valor del radi es pot ajustar per a produir diferents patrons de moviment.

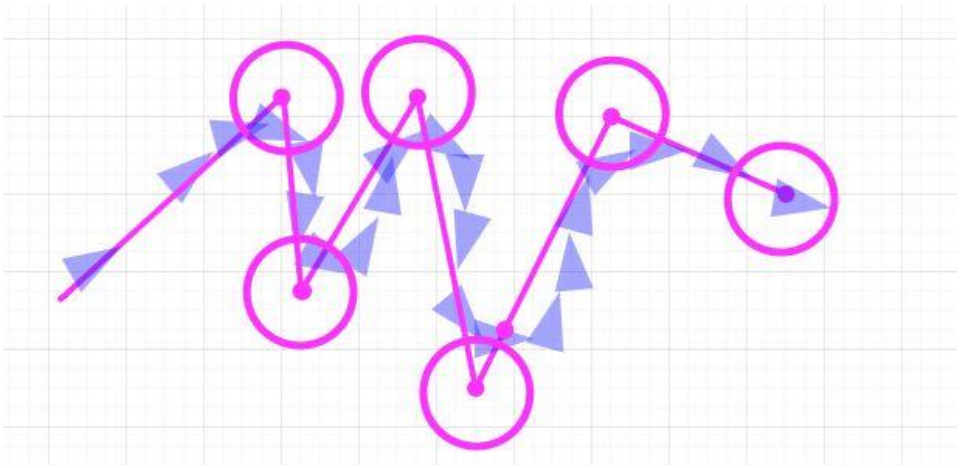


Figura 32: Path Following amb objectius amb àrea

L'última millora que podem afegir és que quan arribi al final del camí, torni enrere per aconseguir que l'agent no es quedi parat al final del camí. Hi ha dues maneres possibles de tornar enrere, que torni sobre els seus passos o que torni al principi del camí com si fos un circuit.

La implementació que s'ha fet és la primera opció que és la més natural, però amb un petit canvi en el codi es podria aconseguir l'altre comportament.

11.3.12 Collision Avoidance i Obstacle Avoidance

En aquest comportament tractarem i implementarem tota la lògica necessària per esquivar obstacles i col·lisions. Quan parlem de col·lisions, ens referim a col·lisions amb altres objectes mòbils, per exemple d'altres agents, per tant quan parlem de *obstacle avoidance*, parlem sobre esquivar col·lisions amb objectes fixes, com obstacles.

Hem d'entendre que aquests tipus de comportaments no són un algorisme de Pathfinding com pot ser l'algorisme de Dijkstra [37] o l'algorisme de A* [38]. Aquests comportaments permetran la navegació en el mapa, esquivar obstacles i eventualment trobar una ruta a través dels blocs, però no funciona bé quan els obstacles tenen forma d'embut, ja que no trobarà la manera de sortir d'allà.

La implementació d'aquests algorismes assumeix que, tant els obstacles com els agents es poden aproximar a esferes, tot i que el concepte basic pot estendre's a d'altres models amb formes més precises. La utilització d'esferes permet una representació eficient donat que no s'utilitzen formes complexes.

Hi ha moltes d'implementar aquests algorismes i nosaltres hem provat de diferents i les hem classificat com a Solució 1, Solució 2, etc.

Solució 1: llençar un raig de mida fixa davant de l'objecte

La primera manera que hem provat es basa en generar una força de *Steering*, com les vistes anteriorment, per esquivar els obstacles cada cop que estan suficientment a prop de l'agent. Inclús si el món té molts obstacles, el comportament utilitzarà un d'ells per a calcular la força.

En aquesta primera implementació, només s'analitzen els personatges que estan al davant, el més proper, aquest l'anomenarem el més perillós. Com a resultat, l'agent serà capaç d'evitar els obstacles en una àrea, navegant pel món amb fluïdesa i sense problemes.

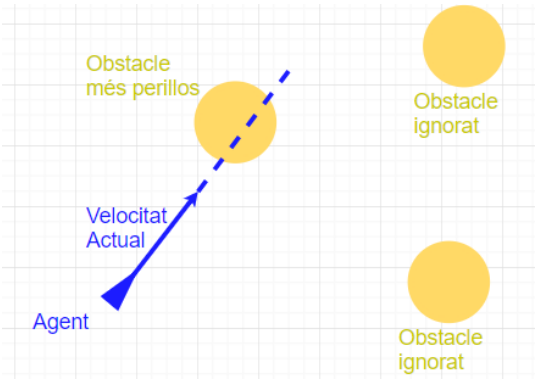


Figura 33: Primera implementació del Collision/Obstacle Avoidance

El primer pas per esquivar obstacles a l'entorn és percebre'ls. Els obstacles pel que s'ha de preocupar són els que té davant seu i bloquegen directament la seva ruta.

Com ja sabem dels comportaments anteriors, el vector de velocitat descriu la direcció de l'agent. Crearem un nou vector que serà una còpia del vector de velocitat però amb diferent longitud. Anomenarem a aquesta component vector de visió i podem definir la longitud màxima d'aquest vector de visió amb una variable anomenada Max_Vision.

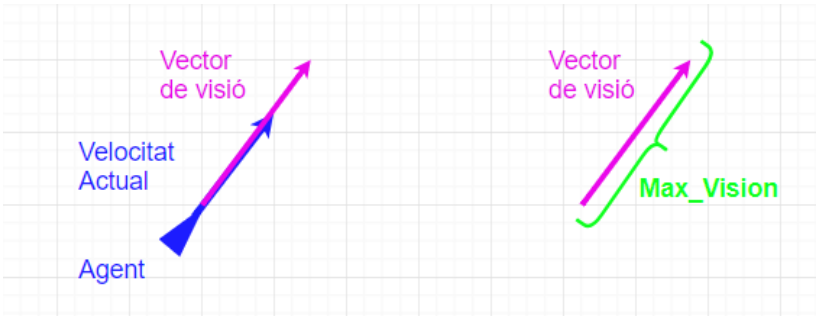


Figura 34: Vector de visió

El vector de visió es calcula com el producte de la velocitat normalitzada i Max_vision i sumant la posició de l'agent en aquell moment. La longitud d'aquest vector determina com de lluny el personatge percep l'entorn. En pseudocodi seria així:

```
Vector3 vision_vector = position + velocity.normalized() * Max_Vision
```

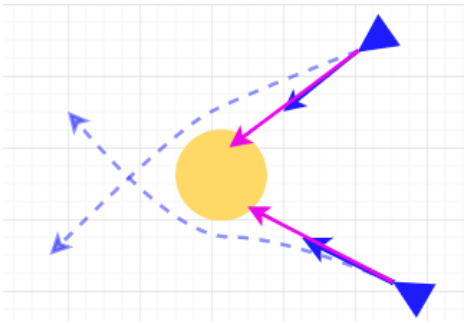


Figura 35: Funcionament de Collision/Obstacle Avoidance amb vector de visió

Una possible manera per a verificar la col·lisió és la intersecció línia-esfera: la línia és el vector avançat i l'esfera és l'obstacle. Aquest enfocament funciona, però té diversos inconvenients donada la senzillesa de l'algorisme. Els veurem més endavant.

Tenim la sort que podem aconseguir aquest comportament d'una manera bastant senzilla atès que les eines que ens proporciona Unity són molt adients per aquesta tasca.

Unity implementa una característica anomenada *RayCast* [39]. Resumidament, aquesta característica permet emetre un raig on les seves components són un punt d'origen, una direcció i una distància màxima. A més, la característica més important d'això és que permet saber si aquest raig ha impactat sobre algun *collider* i també ens permet saber en quin punt ha sigut, és a dir, quin és exactament el punt de col·lisió del raig amb un *collider*.

Com podem veure, això és exactament el que ens cal per al nostre vector de visió, ser conscients que tenim davant, si existeix algun obstacle i si és així, quin és el punt i a quina distància es troba. A més a més, tenim la possibilitat de saber quina és la normal d'aquell punt, component imprescindible per a calcular la força de *Steering* que ens permetrà evitar la col·lisió.

A l'igual que al comportament de separació, aquí també hem d'aplicar una força de repulsió que serà definida als paràmetres de l'agent. A l'igual que abans, ha d'haver-hi un equilibri lògic entre esquivar i el realisme. Aquest valor sempre s'aconsegueix d'una manera empírica

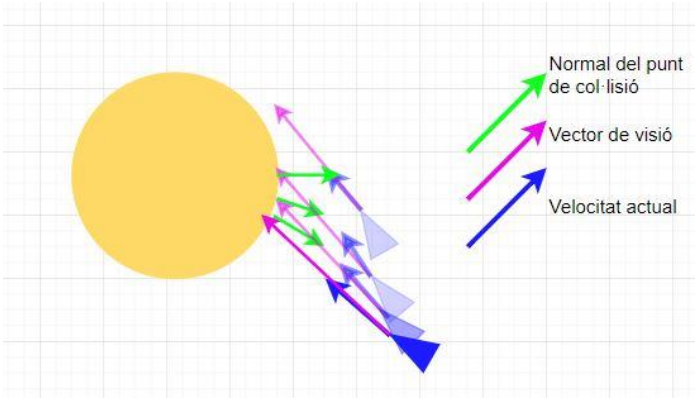


Figura 36: Funcionament de Collision/Obstacle Avoidance amb vector de visió en direrents frames

Com podem veure a la figura anterior, aquella seria la manera en la qual aniria esquivant objectes. Cada cop que el *Raycast* detecta a un *collider*, donat el punt de la col·lisió a la superfície sabríem quina normal té, per tant, amb la força de repulsió abans esmentada i la normal podem aconseguir una força de *Steering* que ens empenyi al nostre agent per esquivar l'objecte.

Aquesta forma funciona i té un alt rendiment per la seva senzillesa, però no és del tot realista, ja que comporta una sèrie d'impediments que ara veurem en detall.

El primer d'ells és que el vector de visió té sempre la mateixa longitud, és a dir, la longitud del vector està definida per la variable Max_Vision, però la longitud del vector hauria de variar amb la velocitat, si no el que pot passar és que tot i anar molt a poc a poc esquivi objectes que estan lluny de l'agent, per tant aquest vector de visió ha de ser dinàmic, tenint en compte la velocitat de l'agent.

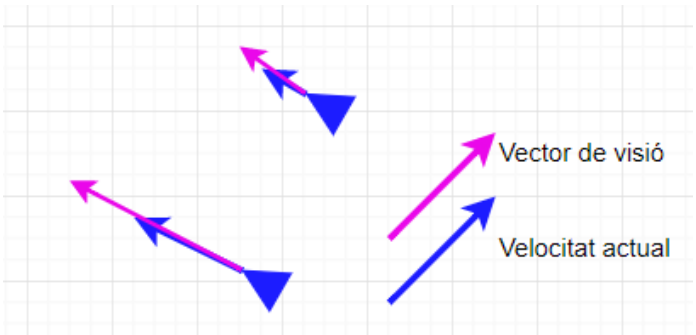


Figura 37: Vector de visió dinàmic segons la velocitat actual

Solució 2: llençar un raig de mida dinàmica davant de l'objecte

Aquest vector dinàmic el podem aconseguir amb un senzill càlcul. Ja que hem de tenir en compte la velocitat, hauríem de crear una nova variable que contingui la magnitud del vector de velocitat dividit entre la velocitat màxima de l'agent. Això ens dóna un valor entre 0 i 1 on 0 és quan l'agent està aturat i 1 quan l'agent va a la màxima velocitat. En pseudocodi quedaria així:

```
Vector3 vision_vector = position + velocity.normalized() * Max_Vision * (velocity.magnitude / Max_Velocity)
```

Amb aquest petit canvi es millora en gran mesura la precisió a l'hora d'esquivar obstacles o d'altres agents, sobretot quan la velocitat és diferent de la velocitat màxima però, tot i que ara aquest comportament és més precís, seguim tenint problemes.

Un altre problema que tenim és que aquest comportament només percep l'entorn dels obstacles que estan justament al davant seu seguint la línia imaginaria que pintaria el vector de visió.

Solució 3: llençar diversos rajos de mida dinàmica davant de l'objecte

Una manera de solucionar això és afegir més vectors de visió al voltant de l'agent. Això es pot fer de moltes maneres i hi ha diversos plantejaments diferents.

Un d'ells és afegir un con de visió en comptes de tenir un sol vector, és a dir, afegir-ne tres vectors i que formin un con.

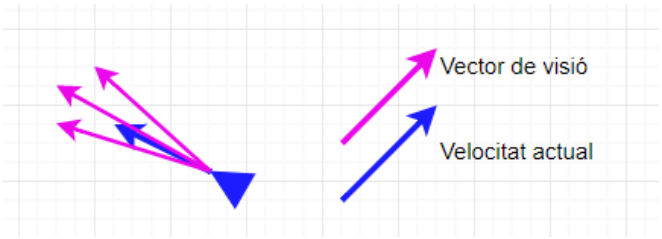


Figura 38: Collision/Obstacle avoidance amb con de visió dinàmic

Això ens soluciona part de la problemàtica derivada de la precisió quan, en la manera anterior, no veuríem perquè no estan situats estrictament davant de l'agent, i amb aquesta manera si que els podem percebre.

La idea ideal és que aquest con de visió sigui aproximadament igual d'ample que l'agent, per determinar si algun dels dos vectors de visió dels laterals col·lisionarien amb algun objecte que estigui situat davant de l'agent. Això ens permet guanyar precisió a l'hora d'esquivar objectes o d'altres agents.

Seria lògic pensar que es poden afegir més vectors de visió al voltant de l'agent per assegurar-nos que l'agent és capaç de repel·lir col·lisions quan vinguin des d'un angle molt major al que tenim amb el con de visió.

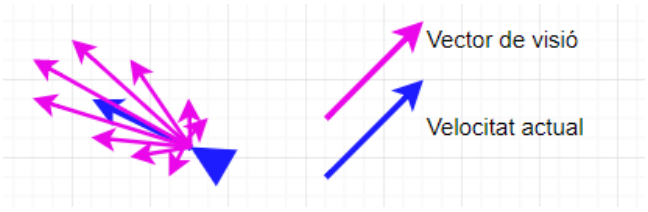


Figura 39: Obstacle/Collision avoidance amb múltiples vectors de visió

Amb aquesta tècnica que és senzilla d'implementar aconseguim un bon comportament a les simulacions. Té una bona percepció de l'entorn i permet esquivar obstacles o agents amb una bona precisió i realisme tot i que no és del tot perfecte perquè sempre esquiva l'objecte més proper. Però té un major inconvenient que fa que no compleixi una premissa bàsica del nostre projecte i aquesta és l'eficiència.

Quan hi ha un nombre de personatges de l'ordre de 15-20, el rendiment és bo, però quan hi ha de l'ordre de 100, que és un nombre interessant per a fer una bona simulació de multituds, el rendiment cau en picat. Això és pel fet que a cada *frame*, cada agent té de l'ordre d'uns 11-15 raycast constantment llençant rajos i comprovant si aquest ha impactat d'alguna manera.

Quan tenim pocs rajos en total el rendiment és bo, però en simulacions grans amb 100 o més personatges amb més de 1300 rajos a cada *frame* (recordem que cada *frame* és cada 0.02s) és massa per a la CPU del nostre computador, no és gens eficient, almenys no pel propòsit general del projecte, per tant s'ha de buscar una altra manera.

Arribats a aquest punt hi ha dues opcions:

- La primera és intentar millorar d'alguna manera aquestes solucions, cosa complicada per la intensiva utilització que estem fent per part dels *raycast* on qualsevol possibilitat de millora de l'eficiència és impossible perquè no es té accés al codi font de la implementació de Unity. Reduint el nombre de rajos a nombres entre 5 o 7, l'eficiència millora però encara hi ha un baix rendiment en altes densitats d'agents, i els comportaments han perdut massa qualitat.
- La segona opció és buscar un enfocament diferents de l'utilitzar amb els rajos, mantenir una idea similar però amb un altre tipus d'implementació.

Afortunadament, Unity és una plataforma molt madura que proveeix moltes eines que, al principi no semblen tenir utilitat aparent però en problemes molts concrets com aquest, es poden utilitzar amb una finalitat diferent de la que potser estan pensats però que a la vegada són molt útils.

Ens estem referint a una característica de Unity que si bé no té nom oficial com a tal, nosaltres l'hem batejat com *Ghost Object.*

Solució 4: utilitzar un Ghost Object com a detector de perills

Definim un Ghost Object com a un objecte de Unity, com els que hem vist abans (obstacles, agents...), on només te la *transform* i un *collider*. Res més. Això vol dir que un *Ghost Object* és un *collider* sense cap objecte físic, per això es diu que és fantasma, perquè aparentment, no té res i per tant tampoc és veu.

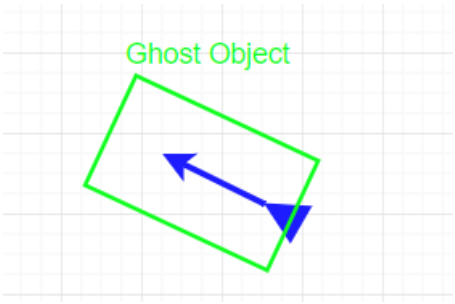


Figura 40: Agent amb Ghost Object

En un primer moment, és evident que tal i com està ara la idea que estem desenvolupada, no ens pot ajudar i s'ha de modificar el comportament, ja que tenim un *collider* que proporciona les qualitats físiques, és a dir, va col·lisionant amb tot allò que es trobi al seu camí.

Afortunadament, Unity ofereix una propietat en els *colliders* que ens pregunta si vol que els fem de tipus *Trigger*, és a dir, desactivar les seves qualitats físiques a canvi de fer que s'activin diversos events relacionats amb les col·lisions.

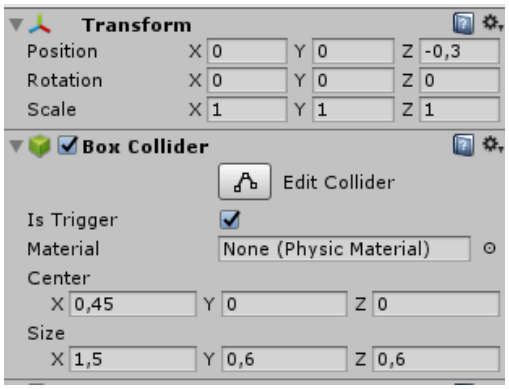


Figura 41: Vista de Unity sobre un Ghost Object

Aquests esdeveniments són:

- **OnTriggerEnter**: s'activa un event que ens indica que un *collider* aliè ha entrat en contacte amb el nostre *collider* del *Ghost Object.*
- **OnTriggerStay:** s'activa quan hi ha un collider dins del nostre Ghost Object.
- **OnTriggerStay:** s'activa quan un collider deixa d'estar dins del nostre Ghost Object.

Veients aquests esdeveniments que ens aporten gran quantitat d'informació que pot ser utilitzada pel nostre comportament i veient també que el *collider* del *Ghost Object* ja no col·lisiona amb altres *colliders*, podem aprofitar aquestes característiques pel nostre comportament.

Utilitzant un collider de tipus caixa com el que hem pogut veure a la figura 39, i els esdeveniments de *OnTrigger*, podem aconseguir un comportament semblant al que teníem amb els *raycasts*.

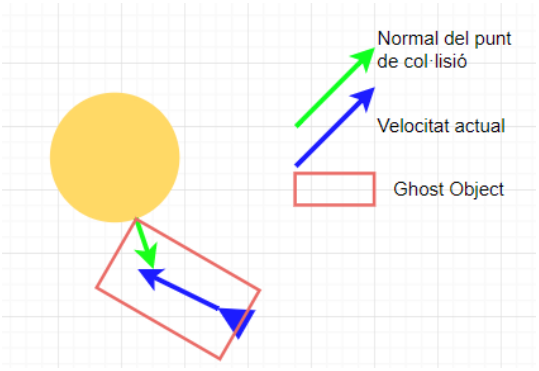


Figura 42: Collision/Obstacle avoidance amb Ghost Object

Una manera d'aconseguir el comportament és, com hem mencionat, utilitzant els events del *collider*. Per simplificar el problema, s'utilitzen només dos dels tres events abans mencionats, el *OnTriggerStay* i el *OnTriggerExit*. El motiu és que el *OnTriggerEnter* i *OnTriggerStay* són molt semblants, només que el *OnTriggerEnter* només salta una vegada, només en entrar, però com ens interessa que salti en entrar i mentre el *collider* aliè estigui dintre, ja ens val amb el comportament del *OnTriggerStay*,

La idea bàsica és saber, en tot moment, quin és l'obstacle més proper per a poder-lo esquivar. Per això ens cal una estructura de dades per a poder emmagatzemar aquesta informació. L'estructura preferida per aquesta tasca és un *HashSet* per l'eficiència que proporciona en les operacions d'inserció, cerca i esborrat. Tot i així, donat que no tindrà un nombre d'elements gran, com a molt tindrem uns 10-15, es podria utilitzar un *ArrayList* també, però amb el *HashSet* podríem escalar millor.

A cada *frame* anem afegint al conjunt els *colliders* nous que detecta el *OnColliderEnter*, només s'afegeixen els nous, així també s'eliminen els *colliders* que ha detectat el *OnTriggerExit*.

Després amb cada un dels *colliders* que tenim al nostre conjunt, es determina quin és el més proper per evitar-lo. Això és senzill, ja que tenim tota la informació del *collider* a la nostra estructura de dades i com a conseqüència d'això, podem obtenir les seves coordenades i el seu radi, per tant podem calcular la *Steering Force* amb aquestes dades. Amb aquesta implementació el rendiment és molt fluid inclús amb 100 agents, per tant és el camí a seguir.

Tanmateix, podem deduir que aquest enfocament (i també en els anteriors) encara no és correcte perquè estem evitant l'obstacle o agent més proper, però més proper no vol dir que hi hagi una col·lisió entre ells, per tant s'ha de pensar com millorar aquest comportament i tenir en compte les trajectòries.

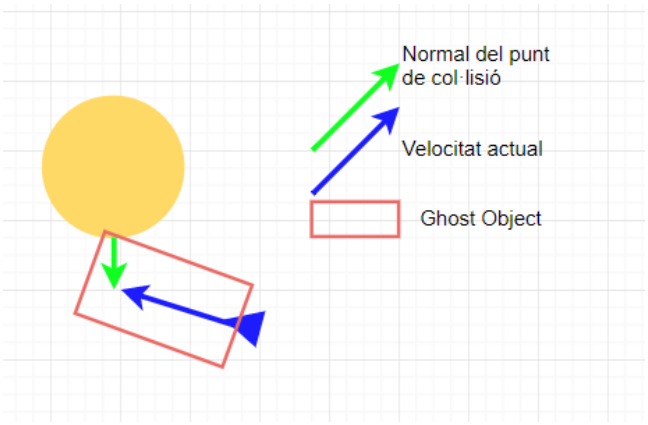


Figura 43: Obstacle/Collision avoidance amb Ghost Object. Esquiva tot i no col·lisionar amb l'agent

Tenim la infraestructura adient, hem vist que el comportament del *ghost object* és bo i el rendiment també ho és, per tant l'últim pas és tenir una simulació de trajectòries per a veure futures col·lisions. Es podria dir que l'idea és semblant al vist al *Follow* on prediem/estimàvem el futur.

Per començant amb la millora del comportament primer hem de definir que volem fer. Principalment, com hem comentat abans, només esquivar si en el futur pròxim hi haurà una col·lisió. Hi ha diferents maneres d'enfocar el problema. En el cas escollit pel projecte, definim l'obstacle o agent més perillós calculant el temps fins a la col·lisió. Explicarem en profunditat aquest concepte a continuació.

Suposem un escenari com el de la figura següent:

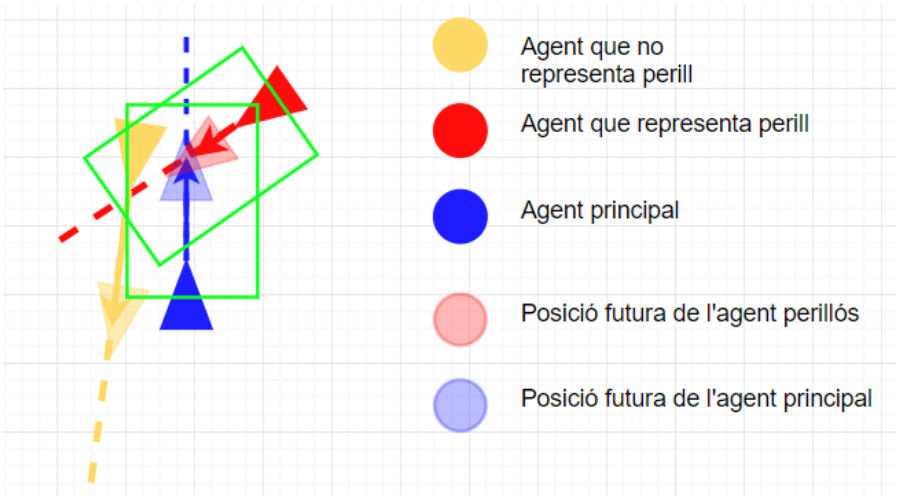


Figura 44: Obstacle/Collision Avoidance amb predicció de trajectòries

Com podem observar, tot i tenir més a prop l'agent de color groc, no representa un perill per a l'agent blau perquè la seva trajectòria no interferirà amb la seva. En canvi, amb l'agent de color vermell si que col·lisionarà amb l'agent blau al futur. Això és degut a la seva velocitat actual, si fos més ràpid, com l'agent groc, o més lent o inclús parat, probablement no hi hauria col·lisió. En el *frame* de la figura, veiem que és l'agent blau qui detecta a l'agent vermell però no a l'inrevés, per tant en aquest *frame*, l'únic que farà alguna cosa per evitar la col·lisió serà l'agent blau.

Pot semblar també que l'agent vermell hauria d'esquivar a l'agent groc però no és així perquè quan l'agent vermell arribi a creuar-se a la trajectòria de l'agent groc, l'agent groc haurà avançat i l'agent vermell quedarà darrere d'aquest, com podem observar veient les posicions futures que representen la posició al següent *frame*, per tant l'agent vermell no ha de fer res en aquest sentit.

Primerament, per implementar aquest comportament per cada agent o obstacle que estigui dins del nostre *ghost object*, ens cal calcular la velocitat relativa i la posició relativa que tenim respecte al *collider* detectat. També ens cal saber la magnitud de la velocitat relativa. Podem veure el pseudocodi següent:

```
Vector3 relativePosition = agent.position - colliderDetectatGhostObject.position;
Vector3 relativeVelocity = agent.velocity - colliderDetectatGhostObject.velocity;
float relativeSpeed = relativeVelocity.magnitude;
```

Aquesta informació la necessitem per a calcular el temps que queda fins a la col·lisió. Si el resultat d'aquesta operació és un valor més petit o igual a 0, vol dir que els dos *colliders* estan col·lisionant en aquell instant. Obtenim de la següent manera aquesta mesura:

```
float timeToCollision = -1 * Vector3.Dot(relativePosition, relativeVelocity) /
(relativeSpeed * relativeSpeed);
```

Sabent el temps que queda fins a la col·lisió, podem predir quina separació tindran els *colliders* en el temps calculat anteriorment. Això implica que si la separació que calculem es inferior a la suma dels radis dels dos *colliders*, voldran dir que s'estan superposant, és a dir, que estaran col·lisionant.

```
Vector3 separation = relativePosition + relativeVelocity * timeToCollision;
float minSeparation = separation.magnitude;
if (minSeparation <= agent.radi + colliderDetectatGhostObject.radi)
```

Si la condició anterior es compleix, tenim una col·lisió futura. En aquest cas, hem de guardar tota aquesta informació perquè aquest procés el repetirem per tots els *colliders* detectats pel *Ghost Object* i ens quedarem amb el més perillós, que com hem comentat abans, és el que tingui el temps menor.

Un cop tenim el *collider* més perillós, és hora d'aplicar la *Steering Force* per evitar la col·lisió. A l'igual que en molts dels altres comportaments anteriors, tenim una força de repulsió que serà introduïda per l'usuari i com sempre, a més gran sigui aquest valor, més repulsió tindran. Com sempre, hem d'aconseguir l'equilibri entre evitar col·lisions i el realisme.

Un dels càlculs necessaris pel càlcul de la força de Steering ja el tenim calculat d'abans, és el contingut de la variable *separation*, només falta normalitzar-ho i fer el producte amb la força de repulsió introduïda per l'usuari.

Ara, com hem fet anteriorment en molts comportaments, hem d'enviar aquesta força que ens ha d'empènyer per evitar l'obstacle. La manera de fer-ho com ja hem pogut veure abans, és enviar aquest vector al comportament de *Seek*, perquè ens enviï cap allà com podem veure en la figura següent:

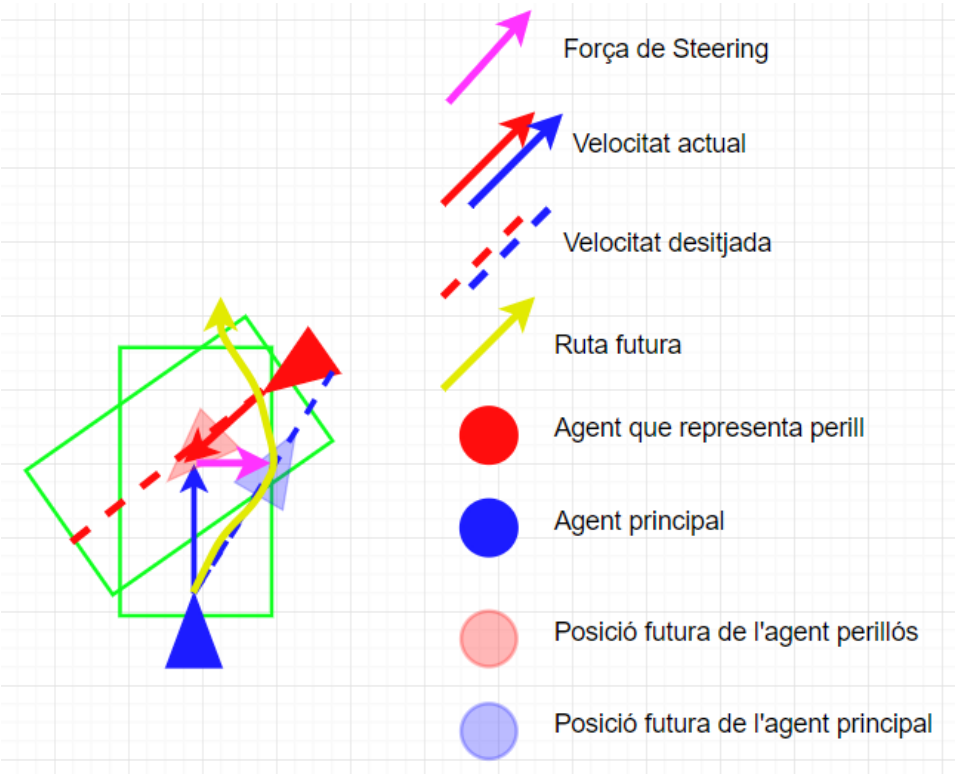


Figura 45: Obstacle/Collision avoidance amb Ghost Object i percepció del futur

Amb aquest comportament implementat, es pot observar que el comportament és el que desitjàvem i té un bon rendiment mentre que el nombre d'agents és alt. Tot i que encara seguim tenint un problema que és inherent al fet d'esquivar obstacles i el propi funcionament dels comportaments de *Steering*. En anglès és conegut com a *Flickering*, parpelleig en català. En el següent apartat veurem que és i com l'hem resolt.

12 Millores implementades

12.1 Eliminació del Flickering

El *flickering* és un efecte no desitjat que apareix en els algorismes que esquiven agents i obstacles però on més patent es veu és a l'hora d'esquivar obstacles, i utilitzen els comportaments de *Steering*.

És un efecte que consisteix en què quan un agent està somes a la força de *Steering* que l'empeny per esquivar un obstacle o agent i està relativament lluny de l'obstacle, pel mateix funcionament del comportament de *Seek*, que sabem que busca el camí mínim entre els dos punts (perquè la velocitat desitjada sempre apunta a l'objectiu), torna a apropar-se a l'obstacle amb la conseqüent repulsió, però després torna a apropar-se a l'obstacle i així, aquesta sèrie de repulsions i apropament fa un efecte de parpelleig, perquè fa moviments molt ràpids entre esquivar i apropar-se.

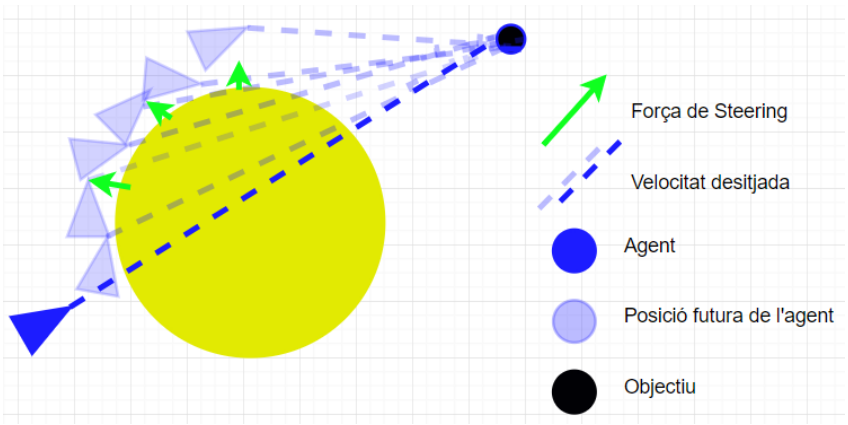


Figura 46: Efecte del Flickering

Tot i no afectar el comportament en si perquè funciona bé igualment, és un efecte molt molest visualment perquè es fa a una freqüència molt alta i sembla que els agents estiguin tremolant a l'esquivar un obstacle o un agent.

Aquest comportament indesitjat succeeix perquè la manera que un agent té d'esquivar objectes es calcula d'una a cada *frame*, sense tenir en compte els moviments anteriors. S'ha de trobar alguna manera de recordar els passos anteriors, tenir certa memòria dels passos previs per evitar tenir comportaments no desitjats com aquests. Aquesta idea, creiem, es pot evolucionar molt per dotar d'un comportament encara més realista, no només per evitar el *flickering*, sinó per altres tipus de situacions on ens caldria saber d'on venim i a on volem anar, sense repetir un i un altre cop els mateixos patrons.

Bé, en el cas del *flickering*, com hem comentat anteriorment, ens cal tenir memòria dels passos anteriors. En aquest cas, la informació que ens cal són les *n* velocitats anteriors. Recordem que el vector de velocitat és un vector de direcció i la seva magnitud (o longitud) és la velocitat en si.

La idea bàsica d'això és tenir un registre de les velocitats en *n frames* anteriors, aquesta component *n* la podrà definir l'usuari, per *suavitzar* els moviments. Quan diem suavitzar ens referim a fer que els moviments que impliquen canvi d'orientació, com quan l'agent està evadint algun altre agent o algun obstacle, no siguin discrets, és a dir, que a cada *frame* no s'apreciï que hi ha una repulsió.

Una manera d'aconseguir aquest objectiu és guardant les velocitats de *n frames* en una estructura de dades de cua i cada *n frames*, fer la mitja de totes les velocitats que teníem acumulades per aconseguir l'efecte de suavitzat a l'hora de canviar moviments. S'escull una cua perquè es l'estructura de dades més natural, ja que es guarda una darrere una altre, però es podria utilitzar una pila o una llista.

Resulta evident doncs que a més gran sigui el valor de *n*, més *frames* ens caldran orientar a l'agent i més suau serà el moviment. Igual que anteriorment amb les forces de repulsió, aquest valor s'ha de testejar bé per tal d'aconseguir un moviment realista, ja que un nombre amb un nombre petit de *frames*, l'agent continuarà existint una petita part de *flickering* però amb masses *frames* el canvi d'orientació de l'agent no anirà acord a la pròpia direcció, és a dir, l'agent es mourà en una direcció però estarà orientat d'una altra manera com podem veure a la següent figura:

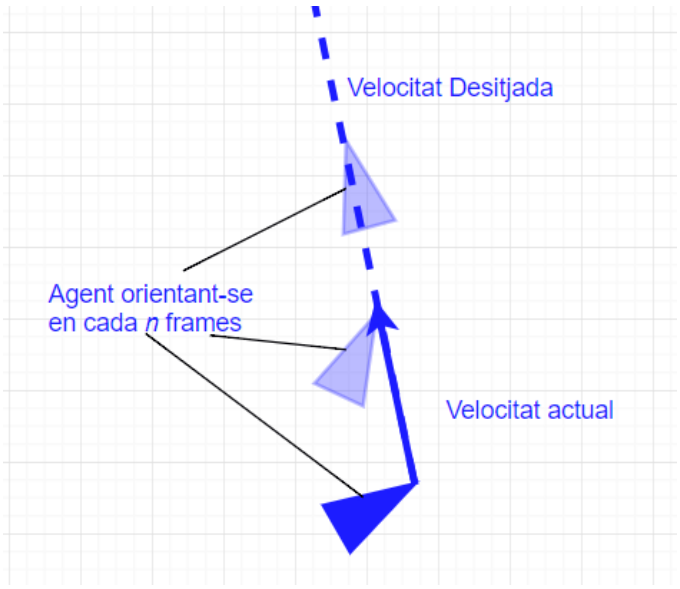


Figura 47: Agent suavitzat amb una component massa gran

12.2 Evasions a baixa velocitat

Una altra millora que hem implementat és el comportament dels agents quan la seva velocitat és baixa.

Un comportament observat era el moviment que tenien en esquivar a baixes velocitats era que els moviments realitzats per aquests eren massa bruscos en comparació a quan anaven a alta o a la màxima velocitat.

Aquest fet és degut a la força de repulsió que l'usuari ha d'introduir, quan la velocitat és baixa, la força introduïda per l'usuari és massa violenta, cosa que no passa quan l'agent viatja a alta velocitat.

Una possible solució és fer que la força de repulsió sigui progressiva, és a dir, que vagi augmentant fins al màxim que és el valor introduït per l'usuari i que vagi disminuint a mesura que l'agent disminueix la seva velocitat.

Això millora molt els comportaments tant d'evasió com quan hi ha una alta densitat d'agents, com quan estan en un embussament, ja que molts cops es veien comportaments semblants al *Flickering* perquè realitzaven petits moviments pràcticament sense moure's, però amb el parpelleig tant característic d'aquest problema.

12.3 Ghost Object variable

Aquesta millora també està relacionada amb el comportament de l'agent a baixa velocitat.

Igual que quan vàrem veure el comportament d'evasió implementat amb *raycasts* que teníem el vector de visió variable que creixia i decreixia progressivament segons la velocitat de l'agent, el *collider* també hauria de fer-ho, tot i que no és tant crític com amb l'algorisme abans mencionat, ja que abans esquivàvem per proximitat i ara per encreuament de trajectòries.

La idea és la mateixa que a l'apartat anterior. L'usuari defineix la mida màxima del *ghost object, per tant* el *collider* va augmentant o disminuint de mida segons la velocitat de l'agent. Això si, hem deixat una mida mínima per evitar comportaments no desitjats com col·lisions per falta de visió.

La millora es veu en l'agilitat que tenen els agents, sobretot amb els obstacles, ja que abans ens succeïa, sobretot al tenir el comportament que l'objectiu fos el clic del ratolí, que a l'estar molt a prop d'un obstacle i fer clic en un altre punt del mapa, els agents donaven un rodeig per anar tot i tenir un camí més curt.

Això era degut al fet que a l'estar tan a prop d'un obstacle i anant a una velocitat baixa, el *ghost object* el detectava dins del *collider* i preveia una possible col·lisió si no donava un rodeig. Ara com la mida és variable respecte a la velocitat, això ja no passa i els agents són molt més àgils que abans.

13 Comparació amb Unity

Unity ens permet implementar, com hem vist anteriorment, molts comportaments que són propis nostres, és a dir, que no venen nativament al motor gràfic.

Tot i així, Unity incorpora nativament dos dels comportaments que hem vist anteriorment. Aquests són el *Seek* i el *Arribe*. Com ja sabem, el *Arribe* és un *Seek* modificat de tal manera que redueix la velocitat quan està a una certa distància de l'objectiu i, Unity implementa exactament aquest comportament.

Si comparem el *Arribe* de Reynolds que hem implementat amb el que implementa nativament Unity, ja podem veure diferencies en el nombre de paràmetres que accepta per modificar el comportament. En el cas del comportament de Reynolds en tenim els de la següent figura:

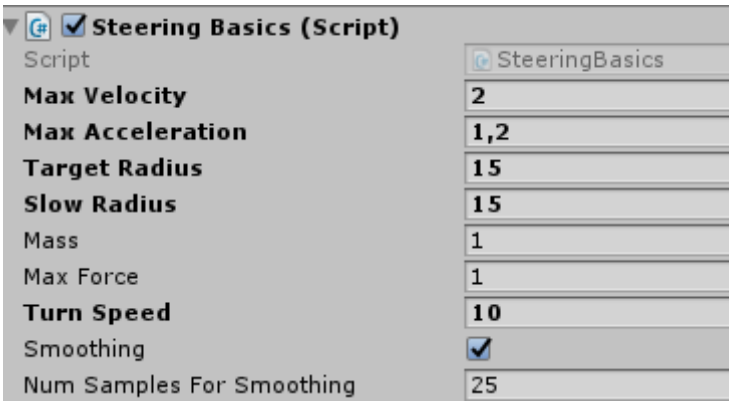


Figura 48: Configuració de Seek/Arribe en un agent de Reynolds

I en el cas dels implementats nativament per Unity:

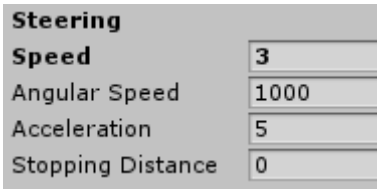


Figura 49: Configuració de Seek/Arribe en un agent de Unity

Els paràmetres que serien comparables son:

- Max Velocity i Speed
- Max Acceleration i Acceleration
- Turn Speed i Angular Speed
- Slow Radius i Stopping Distance

Els paràmetres de la taula anterior són pràcticament iguals en ambdós comportaments, però com podem veure, el nostre comportament implementa molts més i és aquí on radica la diferència.

El A*rribe* de Unity, quan s'apropa a l'objectiu, té una frenada molt brusca que no es pot modificar. Aquesta frenada queda molt diferent de la frenada que s'ha implementat amb Reynolds, que si bé és de per si més suau, es pot modificar jugant amb els paràmetres de la *Mass* i *Max Force* per tenir un ventall de comportaments diferents segons la voluntat del desenvolupador.

També la nostra implementació ofereix quantes *frames* volem agafar per a suavitzar el moviment de l'agent, modificar el valor del paràmetre *Smoothing*, cosa que el de Unity no deixa fer de cap manera. Amb el comportament que implementa Unity, en certes situacions ens trobem amb *Flickering* que no es pot mitigar de cap manera, ja que no implementa cap manera per afrontar aquest problema, en canvi a la nostra implementació, podem variar el nombre de *frames* de per adaptar l'agent a la situació

Unity també incorpora la lògica necessària per a esquivar col·lisions, sigui amb altres agents o amb obstacles. L'únic paràmetre configurable en aquest cas és la "qualitat" a l'hora de la precisió dels moviments mentre esquiva.

Figura 50: Configuració de Obstacle/Agent Avoidance en un agent de Unity

Teòricament, aquesta qualitat indica, com hem comentat, la precisió dels moviments, però aquesta definició és molt ambigua i la documentació oficial de Unity no ens resol la pregunta, no hi ha cap informació al respecte.

L'única manera de testejar el comportament és d'una manera empírica. Tot i així els resultats obtinguts, ni amb *High Quality* que és el valor més alt, han sigut resultats bons, com a mínim en la simulació que nosaltres volem, que és d'almenys 100 agents.

Es poden observar moltes col·lisions, tant entre agents com entre obstacles, a més que com hem comentat anteriorment, hi ha un *Flickering* evident en quant la densitat d'agents és alta i els moviments són a baixa velocitat. Si bé és cert que en l'àmbit del rendiment de la màquina és millor, ho podem associar a què la qualitat del comportament també és bastant baixa.

Això és l'únic que podem comparar actualment, ja que *Unity* no incorpora cap altre comportament comparable als nostres, per tant això realment és un punt molt positiu, ja que, tot i que els comportaments vistos en els anteriors paràgrafs, els que segueixen les directrius de Reynolds són millors, tota la resta no està ni implementat per *Unity*, per tant és un enorme punt a favor als comportaments de Reynolds, ja que ens aporta una gran riquesa i flexibilitat a l'hora de desenvolupar qualsevol cosa amb Unity, sigui un hipotètic joc o, en el nostre cas, la simulació de multituds.

La resta de comportaments que no són d'esquivar col·lisions com el *Flee*, *Pursuit*, etc., es poden arribar a fer, amb *Unity* però. Tot i així, un dels més importants al món de la simulació de multituds i també dels videojocs, és el d'esquivar col·lisions. Aquest, l'única manera de tenir un comportament realment bo és reimplementant el comportament, sigui utilitzant les directrius de Reynolds o ja sigui utilitzant altres maneres, però el que està clar és que per simulacions relativament complexes, jocs amb alta densitat d'agents, etc., el comportament per defecte que incorpora *Unity* no és útil per aquest propòsit.

14 Conclusions

Un cop s'ha finalitzat el projecte, els objectius plantejats en un principi s'han assolit correctament, creant d'aquesta manera una aplicació capaç de recrear una simulació de multituds amb Unity.

En el que respecta a la pròpia simulació, hem aconseguit l'objectiu primordial que vam definir a l'inici del projecte que era millorar el comportament que te Unity davant de certes situacions com tenir un comportament més realista en quan evitar obstacles o col·lisions.

També respecte a la simulació, s'ha aconseguit parametritzar molts dels comportaments implementats per tenir més llibertat a l'hora de configurar la simulació. Tenim molts més paràmetres que els que aporta Unity, fent així les simulacions tant de multituds com de comportaments individuals, més flexibles a les exigències dels usuaris.

D'altra banda, hem aconseguit implementar gran part dels comportaments de l'article de Reynolds, així hem pogut comprovar un gran número de comportaments en diverses situacions, inclús s'ha implementat algun de nou, com el Hide.

S'han pogut realitzar molts experiments variant les dades d'entrada dels scripts per poder provar moltes simulacions diferents per assegurar la qualitat de la solució proposada.

Finalment, ja a un nivell més personal, he descobert la eina Unity i amb això he descobert un món nou que m'apassiona i m'agradaria pròximament seguir profunditzant i guanyar coneixements sobre el món dels videojocs.

15 Treball futur

Tot i que s'han pogut complir tots els principals del projecte, a mesura que s'ha anat avançant, s'ha anat descobrint possibles millores que malauradament, a causa del temps limitat del projecte, no s'han pogut implementar i estaria bé en algun moment veure el seu comportament.

Un exemple d'això pot ser intentar optimitzar el codi eliminant crides de Unity que són costoses, com pot ser calcular la magnitud d'un vector (ja que internament ha de calcular una arrel quadrada i té un cost alt en computació). Hi ha d'altres maneres d'aconseguir resultats similars sense utilitzar algunes crides de Unity.

Hi ha diversos comportaments interessant que no s'han implementat. Entre ells destaca l'algorisme per esquivar parets que, tot i semblar senzill, la idea és complexa, ja que l'agent ha de saber com afrontar una situació amb un mur ample perquè, primer ha de decidir la direcció a la que anar i tot i així, no li garanteix que sigui el camí mínim o que hi hagi final, perquè potser acaba el mapa i el mur segueix.

Seria interessant també incorporar un algoritme de *Pathfinding* per trobar camins entre dos punts. Unity ja incorpora un algorisme d'aquest estil, però està molt lligat al propi moviment i comportaments que nosaltres hem reimplementat, per tant no queda clar com utilitzar la nostra lògica amb tots els comportaments i després utilitzar la part de *Unity* de *Pathfinding*. S'hauria de veure si es pot reutilitzar i si no és així, implementar un algorisme nou.

S'ha intentat implementar, sense èxit, un algorisme per evitar col·lisions directament utilitzant *shaders* a la GPU, però no ha estat possible per culpa de la limitació de temps i a la complexitat del tema. Seria molt interessant veure alguna possible solució utilitzant la GPU, ja que, probablement, el rendiment seria d'alguns ordres de magnitud superior que l'utilitzat a la CPU i es podria fer servir per a simulacions més complexes.

De totes les coses pendents per fer, que n'hi ha alguna més, aquestes són les més destacables. Tot i així, afortunadament són ampliacions del treball, no influeixen en els objectius definits al principi.

Referències

[1] U. Ruelas, «codingornot,» 07 2017. [En línia]. Available: https://codingornot.com/que-es-un-motor-de-videojuegos-game-engine.

[2] «Unity.com,» [En línia]. Available: https://connect.unity.com/p/games-cities-skylines. [Últim accés: 2017 Octubre 23].

[3] «Ign.com,» [En línia]. Available: http://www.ign.com/articles/2016/11/01/super-mario-run-created-with-unity. [Últim accés: 23 Octubre 2017].

[4] «New York Times,» [En línia]. Available: https://www.nytimes.com/2017/05/24/business/dealbook/unity-technologies-400-million-funding-round-pokemon.html. [Últim accés: 23 Octubre 2017].

[5] G. Moore, «Intel,» 19 Abril 1965. [En línia]. Available: https://www.intel.com/pressroom/kits/events/moores_law_40th/index.htm. [Últim accés: 22 09 2017].

[6] Arstechnica, «Arstechnica,» Agost 2017. [En línia]. Available: https://arstechnica.com/gaming/2017/09/build-gather-brawl-repeat-the-history-of-real-time-strategy-games/. [Últim accés: 22 09 2017].

[7] Microsoft, «AgeOfEmpires,» 24 Agost 2017. [En línia]. Available: https://www.ageofempires.com/news/history-of-age-of-empires/. [Últim accés: 22 09 2017].

[8] J. von Neumann, «Wikipedia,» Setembre 2017. [En línia]. Available: https://en.wikipedia.org/wiki/John_von_Neumann. [Últim accés: 22 09 2017].

[9] J. Horton Conway, «Wikipedia,» Setembre 2017. [En línia]. Available: https://en.wikipedia.org/wiki/John_Horton_Conway. [Últim accés: 22 09 2017].

[10] J. Horton Conway, «The fantastic combinations of John Conway's new solitaire game "life",» *Scientific American num 223,* pp. 120-123, Octubre 1970.

[11] A. Turing, «ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM,» 1936.

[12] MIT, «MitPressJournals,» Setembre 2017. [En línia]. Available: http://www.mitpressjournals.org/loi/artl. [Últim accés: 24 Setembre 2017].

[13] R. Sun, Cognition and Multi-Agent Interaction, Març, New York: Cambridge University Press, 2006.

[14] H. Situngkir, Epidemiology Through Cellular Automata: Case of Study Avian Influenza in Indonesia, 2004.

[15] S. Edwards, «The Chaos of Forced Migration: A Modeling Means to an Humanitarian,» p. 168, 2009.

[16] E. Chong, «Epidemiology Through Cellular Automata: Case of Study Avian Influenza in Indonesia,» *Nature-Inspired informatics for Intelligent Applications and Knowledge Discovery: Implications in Business, Science and Engineering,* 2009.

[17] United States Department of Transportation, «Application of Agent Technology to Traffic Simulation,» 2007. [En línia]. Available: https://www.fhwa.dot.gov/research/.

[18] G. Barathy, L. Ylmaz i A. Tolk, «Agent Directed Simulation for Combat Modeling and Distributed Simulation,» de *Engineering Principles of Combat Modeling and Distributed Simulation*, J. W. &. Sons, Ed., 2012, pp. 669-714.

[19] C. W. Reynodls, «Steering Behaviors For Autonomous Characters,» Foster City, California, 1999.

[20] K. Sugeon, S. J. Guy, D. Manocha i M. C. Lin, «Interactive Simulation of Dynamic Crowd Behaviors using General Adaptation Syndrome Theory,» Costa Mesa, California, 2012.

[21] Scrum Alliance, «ScrumAlliance,» Setembre 2017. [En línia]. Available: https://www.scrumalliance.org/why-scrum. [Últim accés: 24 Setembre 2017].

[22] «Team Foundation Server,» Microsoft, 12 2017. [En línia]. Available: https://www.visualstudio.com/es/tfs/.

[23] «Microsoft,» 12 2017. [En línia]. Available: https://docs.microsoft.com/es-es/azure/cloud-services/cloud-services-dotnet-continuous-delivery.

[24] «jobtonic.es,» [En línia]. Available: http://espana.jobtonic.es/salary/26526/16392.html. [Últim accés: 12 Octubre 2017].

[25] «Michael Page,» [En línia]. Available: https://www.michaelpage.es. [Últim accés: 12 Octubre 2017].

[26] «Page Personnel,» 12 Octubre 2017. [En línia]. Available: https://www.pagepersonnel.es/.

[27] «Asset Store,» [En línia]. Available: https://www.assetstore.unity3d.com. [Últim accés: 12 2017].

[28] «Blender,» 12 2017. [En línia]. Available: https://www.blender.org/.

[29] «Maya,» Autodesk, 12 2017. [En línia]. Available: https://www.autodesk.es/products/maya/overview.

[30] «3D Max,» Autodesk, [En línia]. Available: https://www.autodesk.es/products/3ds-max/overview.

[31] «Unity Scripting Reference,» [En línia]. Available: https://docs.unity3d.com/ScriptReference/. [Últim accés: 12 2017].

[32] «Microsoft .Net,» Microsoft, [En línia]. Available: https://www.microsoft.com/net/. [Últim accés: 12 2017].

[33] A. Lian, «Unity,» 07 2017. [En línia]. Available: https://blogs.unity3d.com/es/2017/07/11/introducing-unity-2017/.

[34] «Microsoft,» 03 2012. [En línia]. Available: https://blogs.msdn.microsoft.com/mvpawardprogram/2012/03/26/an-introduction-to-new-features-in-c-5-0/.

[35] «Microsoft,» 09 2016. [En línia]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-6.

[36] «Unity,» 12 2017. [En línia]. Available: https://unity3d.com/es/learn/tutorials/s/roll-ball-tutorial.

[37] «Wikipedia,» 12 2017. [En línia]. Available: https://es.wikipedia.org/wiki/Algoritmo_de_Dijkstra.

[38] «Wikipedia,» 12 2017. [En línia]. Available: https://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda_A*.

[39] «Unity,» Documentation, 12 2017. [En línia]. Available: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html.

[40] Unity, «Unity,» Setembre 2017. [En línia]. Available: https://unity3d.com/es. [Últim accés: 22 09 2017].

[41] J. Snape, S. J. Guy, D. Vembar, A. Lake, M. C. Lin i D. Manocha, «Reciprocal Collision Avoidance and Navigation for Video Games,» Santa Clara, California, 2012.